Data and Container Placement in Scalable Data Analytics Platforms

VORGELEGT VON
M.SC. THOMAS RENNER
GEB. IN QUAKENBRÜCK

VON DER FAKULTÄT IV - ELEKTROTECHNIK UND INFORMATIK
DER TECHNISCHEN UNIVERSITÄT BERLIN
ZUR ERLANGUNG DES AKADEMISCHEN GRADES

DOKTOR DER INGENIEURWISSENSCHAFTEN
-DR. ING. -

GENEHMIGTE DISSERTATION

PROMOTIONSAUSSCHLUSS:

VORSITZENDER:	Prof. Dr. David Bermbach
GUTACHTER:	Prof. Dr. Odej Kao
GUTACHTERIN:	Prof. Dr. Ivona Brandić
GUTACHTER:	Prof. Dr. Tilmann Rabl

TAG DER WISSENSCHAFTLICHTEN AUSSPRACHE: 30. OKTOBER 2018

BERLIN 2018

Acknowledgments

Writing a thesis requires support from various sides. I want to take the opportunity to thank some people who have supported me during that time.

First and foremost, I would like to thank my advisor Odej Kao for his ideas, inspiration, and support since my master's thesis. He provided me a great working environment in his research group over the last years and the opportunities to present my work at international conferences. I also would like to express my appreciation to Ivona Brandić and Tilmann Rabl for their helpful comments and agreeing to review this thesis.

I am thankful to my colleagues and former colleagues at TU Berlin. Especially to Lauritz Thamsen and Andreas Kunft for reading this thesis and giving me valuable input and suggestions. More importantly, we became good friends over the last years, spending a lot of time together in and outside the office.

Furthermore, I would also like to give credit to the former and current members of the CIT team. Especially to Andreas Kliem, Marc Körner, and Alexander Stanik who helped me with my first research efforts. I would also like to thank the students who worked with me during the last years, especially Johannes Müller, Adrian Warszewski, Julian Böhm, and Marius Meldau.

Thanks to Jana Bechstein for proofreading this thesis, helping me with so many administrative issues, and discussions on handling plants and growing vegetables.

Finally, I would like to say thank you to all my family and friends who knew when to ask about my progress and when to distract and support me. Especially, I want to thank Lilian Heim for her patience and support during the last month.

Abstract

Distributed dataflow systems process large volume of data in parallel on multiple machines. In production, multiple dataflow applications are scheduled for execution in virtual containers on a per-job basis. Furthermore, they access datasets partitioned into datablocks across the cluster machines' disks. Runtime performance is important for many of these jobs, as their users expect fast results. However, optimizing performance is difficult, because dataflow jobs are very diverse and used in a wide variety of domains such as relational processing, machine learning, and graph processing. Container and datablock placement decisions impact a job's runtime performance significantly. Furthermore, changing placements affects runtime performance without modifying the application's code, and thus can be applied to many jobs without much configuration effort from the user's side. However, jobs benefit differently from placement decisions, because their resource demands differ from job to job. Hence, there is not a single placement strategy that is optimal for all possible jobs. Besides that, users require a secure long-term data retention for their documents and datasets.

This thesis presents container and datablock placement strategies to optimize the runtime performance of distributed dataflow applications running on shared data analytics platforms. It contributes two placement methods for this. The first method improves the efficiency of a job's dataflow operations and the degree of data locality by colocating its input datablocks and containers on a selected set of nodes. The second method places a job's containers based on network distances between containers and its input datablocks as well as container interference. In addition, this thesis explores the problem of data retention in shared data analytics platforms. Therefore, it contributes a method of storing and accessing lineage metadata through smart-contracts executed on a decentralized blockchain network.

The methods presented in this thesis have been implemented in a research prototype that has been integrated with Hadoop and Ethereum. For evaluation, we used a 64 nodes commodity cluster and workloads consisting of applications implemented in Flink from the domains of relational processing, machine learning, and graph processing. We compared the runtime performance of workloads scheduled with our methods with Hadoop's default placement method. For our blockchain-based data retention method, we measured overhead in terms of additional response time and reported costs using it on Ethereum's blockchain network.

Zusammenfassung

Verteilte Datenflusssysteme ermöglichen, große Datenmengen parallel auf mehreren Rechnern zu verarbeiten. Ressourcen werden ihnen mittels virtueller Container zugewiesen und Eingabedaten werden von den lokalen Festplatten der verteilten Rechner gelesen. Mehrere Datenflussprogramme können zeitgleich auf einem gemeinsamen Rechencluster ausgeführt werden. Schnelle Ergebnisse sind ein wichtiges Kriterium für viele Nutzer. Wesentlichen Einfluss auf die Ausführungsgeschwindigkeit der Datenflussprogramme haben der Ausführungsort der Container sowie der Speicherort der Eingabedatenblöcke. Zudem sind die Datenflussanwendungen vielseitig und unterscheiden sich hinsichtlich ihres Ressourcenbedarfs, wodurch sie unterschiedlich auf ausgewählte Ausführungs- und Speicherorte reagieren. Aus diesem Grund ist es schwierig, eine Platzierungsstrategie zu entwickeln, welche eine optimale Ausführungsgeschwindigkeit für alle Arten von Datenflussanwendungen erzielt. Neben einer schnellen Ausführungsgeschwindigkeit benötigen Nutzer eine langfristige und sichere Speicherung ihrer Daten.

Diese Doktorarbeit präsentiert zwei Datenblock- und Containerplatzierungsmethoden, um die Ausführungsgeschwindigkeit von Datenflussanwendungen zu verbessern. Die erste Methode verbessert die Effizienz der Datenflussoperatoren eines Jobs, indem sie die Eingabedatenblöcke mit Ausführungscontainer auf einer Gruppe von Rechnern kolokalisiert. Die zweite Strategie platziert die Container eines Jobs basierend auf Netzwerk-Hops zwischen Containern und seinen Eingabedatenblöcken sowie Containerinterferenzen. Außerdem erforscht die Arbeit das Problem der Datenspeicherung in geteilten Datenanalyseplattformen durch eine Methode zum Speichern von Linage-Metadaten in einem Blockchain-Netzwerk.

Die vorgestellten Methoden wurden in einem Prototyp implementiert und mit den Open-Source Systemen Hadoop und Ethereum integriert. Die Evaluierung wurde auf einem Commodity Cluster bestehend aus 64 Rechnern ausgeführt. Für die Experimente wurde das verteilte Datenflusssystem Flink verwendet und Programme aus den Domänen relationale Datenverarbeitung, maschinelles Lernen, und verteilte Graphanalyse ausgeführt. Es wurden die Ausführungszeiten der in dieser Arbeit entwickelten Methoden platzierten Anwendungen gegenüber Ausführungszeiten durch Hadoop platzierten Anwendungen verglichen. Für die Blockchain-basierte Langzeitspeicherungsmethode haben wir zusätzliche Antwortzeit, Skalierbarkeit sowie anfallenden Kosten auf Ethereums Blockchain-Netzwerk gemessen.

Contents

Chapter 1: Introduction

Many organizations store and process large volumes of data. Companies like Facebook and Google reported that some of their data analytics applications operate on datasets in the tens of terabytes [1, 2]. Typically, these applications run on dedicated data analytics platforms shared by multiple users and their applications. For instance, CERN runs a data analytics platform for its particle accelerators that subscribes to 20,000 sensors generating two terabytes data per day [3]. Furthermore, its users submit five million analytic jobs daily. Another example affecting our daily lives is given by the Spanish city of Santander, running a smart city deployment with over 15,000 sensors to monitor and analyze the city's environmental conditions [4].

A major challenge is to store and process these large volumes of data at scale in an economically viable way. For this, many distributed storage and processing systems that scale out horizontally using clusters of shared-nothing commodity nodes have been developed by companies, researchers, and open source communities. A driving force for this development has been the high scalability of these systems and the favorable price-performance ratio of using inexpensive commodity hardware, compared to using costly specialized high performance hardware [5].

Many data analytics platforms for batch processing are composed of at least three types of distributed systems [6]. (1) A distributed storage system such as HDFS [7] or Ceph [8] that splits datasets into series of replicated blocks stored across all cluster node's disks. (2) Distributed processing systems such as Spark [9] or Flink [10] that access and process these datasets in parallel on multiple nodes. (3) A resource management system such as YARN [11] or Mesos [12] that provides distributed processing systems access to the cluster's compute resources through virtual containers. This platform design allows to run multiple data analytics applications simultaneously, sharing the resources of a single cluster infrastructure.

One advantage of this shared platform design is the support of a broad range of applications. This is because the resource management and distributed storage systems are application-agnostic and, thus, support multiple distributed processing systems focusing on different types of applications. Furthermore, processing systems like Flink and Spark provide a unified engine for batch and stream processing and include libraries for machine learning and graph processing. Workload traces from production clusters imply that these platforms scale up to more than thousands of applications [13, 14]. At this scale, automation is required, because management and configuration of the platform itself and its large variety of applications goes beyond the capabilities of human administrators. At a first glance, this seems to be important only for a few organizations that operate large-scale deployments. However, even start-ups with smaller cloud deployments and less applications face cost pressure and benefit from platform automation by increasing the utilization of their resources, reducing maintenance costs, and increasing runtime performance [15].

Runtime performance of data analytics applications is important as it can improve the usability of a service due to faster results [16, 17]. Twitter, for instance, promises its users that they update the indexes for their search completion on terabytes of data within ten minutes to always provide relevant results [18]. However, runtime performance in shared data analytics platforms requires a good coordination between task, data and container placement. Poor placement decisions, for instance when placing processing tasks on over-utilized nodes or far away from other connected tasks, can degrade performance [19]. In situations with high cluster utilization, antagonistic workloads can occur in which applications and container combinations interfere with each other, decreasing the runtime performance [20]. Furthermore, jobs benefit differently from placement decisions, because their resource demands differ from job to job. For instance, some tend to benefit more from placing containers close to each other, reducing the amount of intermediate data send through the network between their tasks. While others tend to benefit more from colocating containers and datablocks, improving the degree of data locality.

Furthermore, some data needs to be stored safely and reliably for a long period of time. Reason for a growing demand for long-term storage solutions is a trend towards more organization policies, government laws, and regulations for retaining data. For instance, some companies are required to keep tax information, contracts, and business reports for up to ten years depending on the country they operate in [21]. Another example concerns research data and results that should be publicly

available for a long period of time [22]. These policies result not only in more data that needs to be archived. Moreover, ensuring it's integrity and longevity becomes an important task as well [23]. For instance, proving that data stored long time ago has not been changed or, if it did to track when a file has changed by whom.

1.1 Problem Definition

The topic of this thesis is the design of datablock and container placement methods as well as a long-term data retention method for shared data analytics platforms. The research question of this thesis is:

"How to optimize the runtime performance of distributed dataflow applications and provide long-term data retention on shared data analytics platforms?"

The problem embodied in this question and addressed in this thesis is twofold:

Runtime Performance: Data and container placement decisions are major factors for runtime performance of distributed data processing applications in shared data analytics platforms. The first part of the problem therefore asks how runtime performance can be optimized by a better coordination between datablock and container placement decisions without allocating more resources.

Data Retention: Data integrity and longevity are important to ensure long-term data retention in shared data analytics platforms. The second part of the problem therefore asks how long-term storage can be improved by storing and accessing necessary lineage metadata through a blockchain network.

The proposed solution of this thesis is a platform that automatically selects an appropriate datablock and container placement strategy for storing a dataset or executing an application. Besides a system architecture, we present two methods that optimize runtime performance through better coordination between application containers and datablock placement decisions. A third method presents a decentralized approach to support long-term data retention in this platform, which runs on a peer-to-peer blockchain network.

We make the following three assumptions while addressing the problem:

Shared Data Analytics Clusters We assume that applications are executed on commodity clusters, in which they get a cluster share in form of virtual containers and access data from a colocated distributed file systems. Furthermore, we assume control over the placement of these containers and datablocks.

Distributed Dataflow Systems We assume workloads that consists of distributed dataflow applications with task executed in virtual containers.

Batch Processing Workloads We assume mixed batch workloads that consists of iterative machine learning programs, graph processing and data-intensive relational database queries.

We use standardized benchmarks and algorithms used in different domains for evaluation. In addition, we do not modify the distributed dataflow system. Instead, we treat it as black box that is executed in virtualized containers and optimize its runtime performance by only changing its container and datablock locations.

1.2 Contributions

This thesis proposes a set of solutions to the previously described challenges. These solutions make contributions in three areas.

The first area of contribution covers two placement methods to optimize the runtime performance of distributed dataflow applications in shared data analytics platforms. The first placement method is called *CoLoc* and focuses on data-intensive recurring applications. It consists of a data and a container placement algorithm that colocates a job's input datablocks and containers on the same set or subset of pre-selected nodes. As a result, these applications benefit from a high degree of data locality and more local inter-process communication between containers. The second placement method is called *NeAwa* and is using network metrics for making placement decisions. It takes network distances between all involved containers as well as between containers and input datablocks into account when placing containers. Furthermore, it balances containers on selected nodes to avoid interference with each other. NeAwa's placement algorithm is based on a cost function reflecting these

objectives and it uses Simulated Annealing (SA) for approximating a placement with the lowest costs.

The second area of contribution explores the problem of long-term data retention in shared data analytics platforms. For this, a framework called *Endolith* is presented that uses a smart contract-based blockchain to improve long-term storage. In this approach metadata that describes the lineage of data transformed by the platform is stored immutable on a blockchain network. These metadata is only accessible through well-defined functions that are invariant as they are deployed and executed as smart contracts on the blockchain. As a result, it is possible to store datasets and results reliably for a long-term, without relying on a central trust authority.

The third area of contribution is given by the architecture of a data and resource management system to improve runtime performance and data retention in data analytics platforms. This is done by automatically selecting an appropriate placement strategy per dataset and application. Emphasis will be given on the strategy selection mechanism and integration with related state of art systems.

We have implemented all methods in a research prototype system that is integrated with YARN as resource management system, HDFS as distributed file system and Ethereum as smart contract-based blockchain. We use Flink as reference distributed dataflow system for our evaluation and different types of batch applications. We evaluated all three prototypes with several application workloads on a 64 worker nodes commodity cluster and a 8 worker node fat-tree testbed at TU Berlin.

Main contributions of this thesis have been published as follows:

1. **Thomas Renner**, Johannes Müller and Odej Kao. *Endolith: A Blockchain-based Framework to Enhance Data Retention in Cloud Storages.* In the Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDB). IEEE. 2018
2. **Thomas Renner**, Lauritz Thamsen and Odej Kao. *Adaptive Resource Management for Distributed Data Analytics based on Container-level Cluster Monitoring.* In the Proceedings of the 6th International Conference on Data Science, Technology and Applications (DATA). SciTePress. 2017.
3. **Thomas Renner**, Lauritz Thamsen and Odej Kao. *CoLoc: Distributed Data and Container Colocation for Data-Intensive Applications.* In the

Proceedings of the 2016 International Conference on Big Data (BigData). IEEE. 2016.

4. **Thomas Renner**, Lauritz Thamsen and Odej Kao. *Network-Aware Resource Management for Scalable Data Analytics Frameworks.* In the Proceedings of the 2015 International Conference on BigData (BigData). IEEE. 2015.

The following publications are related to this thesis:

1. Lauritz Thamsen, Ilya Verbitskiy, Jossekin Beilharz, **Thomas Renner**, Andreas Polze and Odej Kao. *Ellis: Dynamically Scaling Distributed Dataflows to Meet Runtime Targets.* In the Proceedings of the 9th International Conference on Cloud Computing Technology and Science (CloudCom). IEEE. 2017.

2. Lauritz Thamsen, Benjamin Rabier, Florian Schmidt, **Thomas Renner** and Odej Kao. *Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference.* In the Proceedings of the 6th International Congress on Big Data (Big Data Congress). IEEE. 2017.

3. Lauritz Thamsen, **Thomas Renner**, Ilya Verbitskiy and Odej Kao. *Adaptive Resource Management for Distributed Data Analytics.* In Advances in Parallel Computing. IOS Press. 2017.

4. **Thomas Renner**, Johannes Müller, Lauritz Thamsen and Odej Kao. *Addressing Hadoop's Small File Problem With an Appendable Archive File Format.* In the Proceedings of the 2017 Computing Frontiers Conference (CF). ACM. 2017.

5. Lauritz Thamsen, **Thomas Renner**, Marvin Byfeld, Markus Paeschke, Daniel Schröder and Felix Böhm. *Visually Programming Dataflows for Distributed Data Analytics.* In the Proceedings of the 2016 International Conference on Big Data (BigData). IEEE. 2016.

6. Lauritz Thamsen, Ilya Verbitskiy, Florian Schmidt, **Thomas Renner** and Odej Kao. *Selecting Resources for Distributed Dataflow Systems According to Runtime Targets.* In the Proceedings of the 35th International Performance Computing and Communications Conference (IPCCC). IEEE, 2016.

7. Lauritz Thamsen, **Thomas Renner** and Odej Kao. *Continuously Improving the Resource Utilization of Iterative Parallel Dataflows.* In the Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE. 2016.

8. **Thomas Renner**, Marius Meldau and Andreas Kliem. *Towards Container-Based Resource Management for the Internet of Things.* In the Proceedings of the International Conference on Software Networking (ICSN). IEEE. 2016.

9. Tobias Herb, Lauritz Thamsen, **Thomas Renner** and Odej Kao. *Aura: A Flexible Dataflow Engine for Scalable Data Processing.* In the Proceedings of the 9th International Workshop on Parallel Tools for High Performance Computing. Springer. 2016.

1.3 Outline of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2: Background presents the necessary background on related data analytics systems and concepts. First, we present the concepts of distributed data processing focusing on distributed dataflow systems. Afterwards, a typical data analytics setup is presented with focus on distributed file systems and cluster resource management systems. We also describe the concept and functionality of smart contract-based blockchain networks, which we use for our decentralized data retention method.

Chapter 3: Related Work first presents related distributed systems used in data analytics platforms. In particular, these are distributed dataflow systems, distributed file systems and resource management systems. Afterwards, related task, container and data placement strategies and schedulers are presented. Finally, related work on Blockchains like Ethereum and alternative implementations as well as Blockchain-based data retention approaches are presented.

Chapter 4: Problem and Concepts presents the problem we address with this thesis. Therefore, we introduce the state of the art and its limitations. Based on this, we introduce our approach and its methods to tackle selected challenges. At the end of this chapter, we discuss assumptions and requirements.

Chapter 5: Data and Container Colocation Placement presents our work on data and container colocation in detail. First, the concept of data and container colocation is discussed. Afterwards, its placement process and algorithms are presented. Finally, an evaluation based on a 64-nodes commodity cluster and different workloads is presented.

Chapter 6: Network-Aware Container Placement presents a container placement strategy based on a cost function that takes possible container and datablock locations as well as load balancing into account. First, the method behind our approach is discussed in detail. Afterwards, the SA-based placement algorithm is presented. Finally, an evaluation based on different workloads on a fat-tree testbed is presented.

Chapter 7: Data Retention Placement presents the concept of our decentralized data retention placement approach. First, we describe the motivation and design principles of our smart contract-based blockchain approach for long-term data retention. Afterwards, we present the workflow and implementation of the approach in more detail. Finally, we present an evaluation of the prototype using HDFS as distributed file system and Ethereum's official testnet as blockchain network.

Chapter 8: Conclusion concludes this thesis by summarizing our results and identifying directions for future work.

Chapter 2: Background

Contents

Based on the problem definition, this chapter provides an introduction to scalable data analytics systems and concepts. Afterwards, the design of a data analytics platform widely used in research and industry is presented. Finally, fundamentals of smart contract-based blockchains are introduced, which we applied to data analytics platforms for improving data retention.

2.1 Scalable Data Analytics Concepts and Systems

Scalable data analytics solve problems that involve large amounts of data and computation. For this, three types of systems are often used in conjunction: distributed processing systems, resource management systems and distributed file systems [6]. All systems have in common that they are highly scalable and designed for clusters

built from commodity hardware, from which they gained a lot of popularity in industry and research due to its favorable price-performance ratio.

2.1.1 Distributed Dataflow Systems

Distributed dataflow systems such as MapReduce [24], Spark [9], and Flink [10] are a class of distributed processing systems that solve analytic tasks by using the concept of data parallelism. In regard to this, the data to be processed is split into multiple partitions and each one is distributed across different nodes that execute the same task on their partition in parallel. Furthermore, all nodes are synchronized and exchange data when the parallel processed data must be combined and merged. Beside data parallelism, the concept of pipeline parallelism is used when multiple tasks depend on each other and can interleave. In this case, the output of one task is streamed as input to the next task.

From the user's perspective, dataflow systems allow to develop scalable data-parallel applications from sequential building blocks. Developers can choose from a set of predefined dataflow operators such as map, reduce and join. Furthermore, they can add individual sequential code as User-Defined Functions (UDF) to the second-order functions map and reduce. Other specific variants of map and reduce are filter and pre-defined aggregations for computing like sums and counts. A join operator combines two dataflows into one by a user-defined join criteria. Furthermore, an Iteration operator allows to execute the same UDF multiple times until a user-defined number of iterations or other criterion has been reached. It is important to emphasize that each operator receives an input and provides an output. As shown in Figure 2.1, operators are connected as a dataflow job graph. In the shown sample graph, two different datasets are read-in and pre-processed by two different operators, afterwards, both flows are joined, reduced and stored.

Figure 2.1: A dataflow graph with multiple connected operators.

Internally, these systems distribute and parallelize the execution of the dataflow graph automatically. Therefore, multiple data-parallel task instances of each operator

are generated. The number of parallel instances is called Degree of Parallelism (DoP) and must be set by the user manually. This can either be done system wide for all operators via a global configuration file or explicitly for each individual operator in the job's source code. As tasks are data-parallel, each task instance receives a partition of the input dataflow, performs the operator's UDF on it and produces a partition of the output dataflow. Partition can either be created by reading parts of the input file from a storage system or it can be received from a predecessor task instance. Moreover, the locality of input data partitions and tasks plays an important role in the performance. This is because tasks are scheduled with priority on nodes storing it's input data in order to access it from local disks. Figure 2.2 parallelizes the previously defined dataflow graph of Figure 2.1 with a DoP of two.

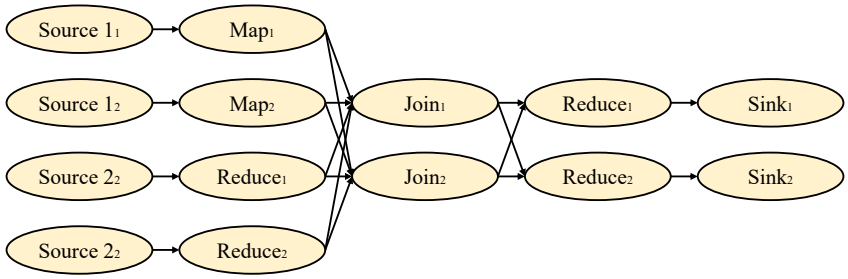


Figure 2.2: A dataflow graph parallelized with a DoP of two.

From an architectural design perspective, distributed dataflow systems often follow a master-worker pattern. One single master manages multiple interconnected workers, on which the task execution takes place. Typically, they assume homogeneous capacities on all workers, so that each worker offers the same amount of cores and memory available for task processing. Task instances are scheduled and executed on all available workers. Each worker provides execution slots that represent the compute units. The number of slots per node is per default equal to its numbers of CPU cores. Moreover, a slot can execute a task or a chain of tasks. Task chains can be executed in parallel by adding pipeline parallelism. Expect for pipeline breaking operators such as joins and iterations that require all elements with certain keys to be available before they start.

Especially, joins and group-based aggregations can be network intensive, as they require all elements of the same group or with the identical join key to be available at the same task instance. Therefore, if the data is not already partitioned by these keys, the dataflow needs to be shuffled. By this, all elements with the same key need to

be moved to the same task instance, leading to all-to-all communication. Moreover, some operators have multiple implementation strategies, which require different synchronization and communication. For instance, if two dataflows are joined, a Broadcast-Forward or Repartition-Repartition join strategy can be applied [25]. The Broadcast-Forward strategy shuffles one dataflow to all parallel task instances, allowing the other dataflow to be pipelined without any shuffling. This strategy is selected when one dataflow is smaller than the other one. The Repartition-Repartition strategy involves shuffling both dataflows, so all elements with the same key are received by the same task instances. Such strategies are automatically selected by the dataflow systems optimizer.

Distributed dataflow systems also support the concept of iterations. They are for instance used in machine learning algorithms such as K-Means or Stochastic Gradient Descent and graph processing algorithms like Page Rank or Connected Components [26]. Figure 2.3 illustrates the concept of iterations in dataflow systems. In each iteration, the same step function that consists of an arbitrary dataflow of operators will be executed multiple times until a termination criterion has been reached. This can either be a maximum number of iterations or a custom aggregators and convergence criterion. The first execution of a step function execution consumes the entire iteration input, computes the next version of the partial solution which in turn is the input for the next step function execution. It is important to emphasize that step functions are executed data-parallel. That is, multiple instances of a step function are executed in parallel on different data partitions. In order to realize synchronization, all parallel step functions must complete before the next iteration will start. Therefore, a synchronization barrier exists between any step function execution. The termination criteria will also be evaluated at these synchronization barriers. Furthermore, iterations can be network-intensive as data needs to be shuffled and repartitioned before a next step function execution starts.

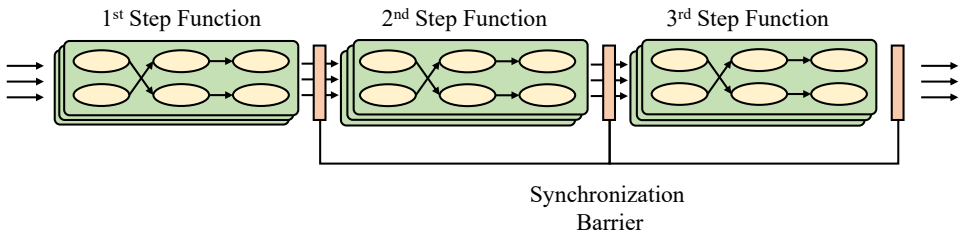


Figure 2.3: An iterative dataflow graph with three step functions.

2.1.2 Resource Management Systems

Data analytics platforms run a diverse mix of data processing frameworks and applications supporting batch, stream and iterative processing [19]. Instead of running multiple dedicated clusters for each framework, data analytics applications are typically executed on top of a resource management system like YARN [11], Mesos [12] or Firmament [27]. These systems allow to execute different applications in parallel by hosting its tasks in virtual containers on multiple nodes. Furthermore, they allow to share resources of a single cluster efficiency among multiple users. From the user's perspective, data scientists also have more freedom to choose the most appropriate frameworks for their analysis task at hand.

Applications in resource management systems are executed on a per-job basis. Each job gets a share of the cluster for its execution through requesting and allocating multiple containers. These containers are allocated on nodes with sufficient space and released when the job execution is finished. Thus, depending on the size of containers and node capabilities, multiple containers can run on a single node utilizing its resources. In comparison to using Virtual Machines (VMs), operating-system-level virtualization by using containers allows allocating and releasing resources faster. Moreover, containers have less resource overhead, as they do not need a separate operating system per container. Another aspect is that most resource management systems use containers without strict resource isolation. Therefore, they can get more resources than beforehand requested and multiple containers hosted on a node struggle for its shared disk and CPU resources. Jobs often have different dominant resources phases and its resource demands fluctuate over time. Therefore, they benefit from statistical multiplexing and allocate free resources that are not used by other colocated jobs at a moment [28, 29].

Figure 2.4 shows the design of a centralized resource management system based on YARN [11] following a master-worker pattern. A single master called Resource Manager receives job submissions and is responsible for scheduling and tracking its statuses. On each worker node, a Node Manager starts, monitors ands releases containers and manages its own resources. Furthermore, one container per job, which is called Application Master, hosts the master of the particular distributed processing system that in turn manages all its workers hosted in further containers, and negotiates resources from the Resource Manager.

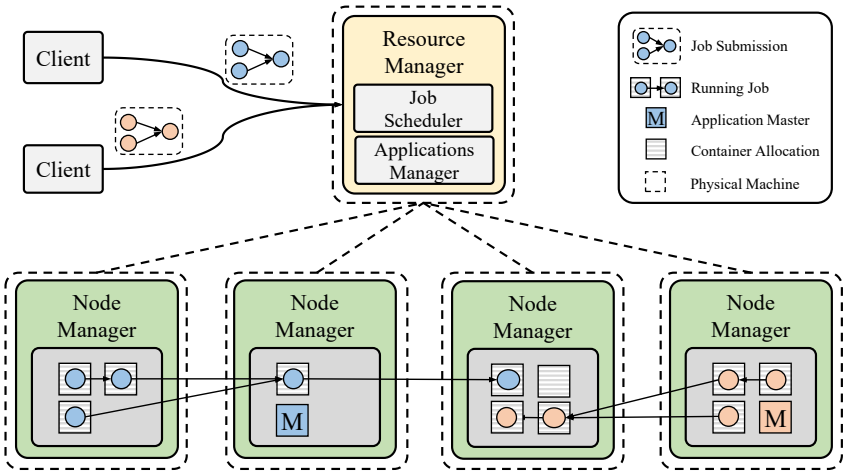


Figure 2.4: A resource management system hosting two different data analytics jobs in multiple containers.

A major component of the Resource Manager is scheduling job containers and making the decision on which node to place them. It performs this scheduling function based on the resource requirements of the job in terms of number of containers and its size, which incorporates CPU cores and memory. Furthermore, many systems allow to use different schedulers focusing on different goals like fairness [30], throughput [31], data locality [32], or deadlines completion [33]. Beside scheduling, the Resource Manager is responsible for managing applications by tracking their status, negotiating the first container of an application and providing fault tolerance features such as restarting containers on failures.

The scheduler in resource management systems often follows a monolithic or two-level approach. Monolithic schedulers [27, 34, 35] are composed of a single scheduling component that handles all job submissions. Also, they place all containers with the same logic, which has lead to some sophisticated schedulers being developed. For example, machine learning-based approaches that avoid bad interference between jobs competing for resources [20, 36]. In two-level schedulers [11, 12] resource allocation and task placement are separated. Therefore, each framework implements its own placement logic to request containers from a central resource manager. For both, the Resource Manager needs a global view of the available resources and running applications. Beyond this centralized design, decentralized approaches exist [13, 37], in which multiple Resource Manager processes are used for increasing scalability and fault tolerance.

2.1.3 Distributed File Systems

Datasets and analytic results in data analytics platforms are often stored in a shared storage system such as a distributed file system. The concept behind these systems is to split files into multiple smaller blocks with a fixed size. These blocks are replicated and distributed on the disks of multiple nodes. This leads to high scalability and read performance, because a file's datablocks can be read in parallel from multiple node's disks by a user's file system client or data analytics application. Moreover, datablock redundancy provides fault tolerance as blocks can be recovered and it allows to continue working when a node stops unexpectedly or a block becomes corrupted. Furthermore, the system can scale-out easily by adding new nodes to the cluster.

Figure 2.5 illustrates the design of a distributed file system based on HDFS [7]. It follows a master-worker architecture. The master, called Name Node, stores metadata that represents the structure of directories and files in a tree. This metadata also covers attributes of files and directories, such as ownership, creation time, permissions and replication factor. Furthermore, the Name Node tracks where across the worker nodes, called Data Node, a file's datablocks are stored. When a client or data analytics application requests a file from the Name Node, it returns the block locations, which the client consequently uses to read the actual file blocks directly from the workers using TCP/IP sockets.

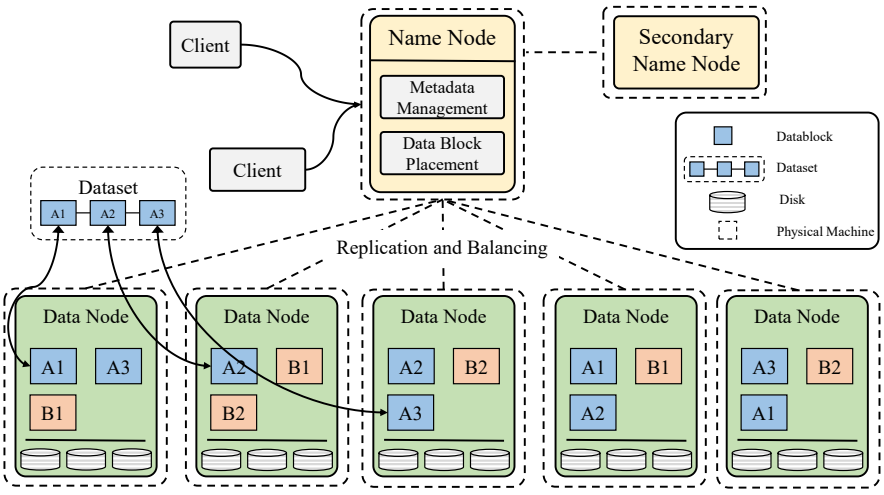


Figure 2.5: A distributed file system storing two different datasets that are partitioned in datablock across the cluster nodes' local disks.

The Name Node is also responsible for datablock placement. When a user or data-parallel processing system writes a file or output to the distributed file system, it writes blocks simultaneously on a number of Data Nodes. Therefore, write performance is low compared to read performance. A common strategy to write new datablocks is to highly distribute a block's replicas among different nodes and racks, mainly due to fault tolerance and balancing reasons. HDFS, for instance, uses a rack aware data placement strategy that assumes three replicas per block, the first block is stored locally, if possible, the second copy on a node of the same rack, and a third one on a node of a different rack.

Distributed file systems specialized for data analytics such as HDFS focus on storing and processing large files in the range of gigabytes to terabytes. Therefore, they are optimized for large files, and provide high aggregate bandwidth and capacity. Under the hood, they use a large default block size of 128 MB. The main reason for this is due to the commonly applied data locality-based task scheduling of the colocated dataflow systems, which schedule its data-parallel task with priority on nodes storing a file's datablock. By this, the task function can be executed on the same node on which the input data is stored without additional network overhead, as a block must not be transferred from another node. Other types of storage system such as Storage Area Networks, Network Attached Storages or cloud-based storages like S3 are also supported by many distributed processing frameworks. However, when using these types of systems, data storage and processing is separated from each other, thus data locality cannot be archived.

2.1.4 Data Analytics Cluster Setup

Data analytics platforms run multiple distributed systems for storing and processing large datasets in conjunction. Figure 2.6 shows the setup of such a platform, which is also recommended by Hadoop and widely used in research and industry [6]. A resource management system is typically colocated with the distributed file system. Jobs of different data processing frameworks run in temporarily reserved containers on top of the resource management system. When a job is submitted, multiple containers are allocated. Afterwards, the required data processing framework is bootstrapped into these containers and the job starts. The containers are released when the job is finished. While running, they access datasets from the colocated distributed file system and also store their results there. Due to the colocation, data

analytics frameworks can schedule a job's task directly on nodes storing the input data, which reduces the network overhead and increases runtime performance.

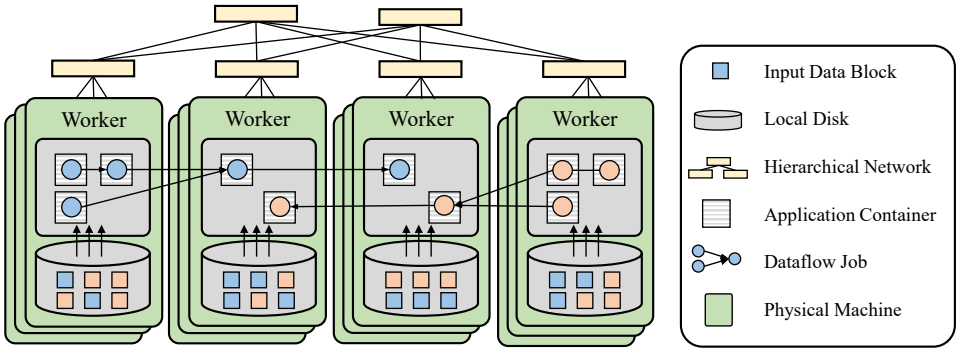


Figure 2.6: A data analytics cluster setup running multiple applications using a colocated resource management system and distributed file system.

Unlike High Performance Computing (HPC) environments, where compute and storage are separated from each other and dedicated nodes are interconnected with high speed links like InfiniBand [38], each node offers its compute and storage capacity for storing and processing data and is equipped with commodity hardware and interconnected by Ethernet. Moreover, the network is organized in a hierarchical topology built with commodity hardware [39]. Nodes are grouped into racks at the lowest level and multiple paths with different hop counts can exist between two nodes of different racks. However, bandwidth between nodes within a rack is higher than between nodes in different racks, because switches are often oversubscribed due to cost saving and maintenance reasons [32, 40]. At the same time, performance of storage mediums like Solid-state drives (SSDs) increase and become more popular in data analytics clusters [41–43]. Also, in-memory storages, like Alluxio, formally known as Tachyon [44], become more popular.

Workload traces from Microsoft with over 20,000 machines [13] and Google [14] with over 12,000 machines imply that productive cluster setups can scale up to more than tens of thousands of nodes and applications. At this scale, management and configuration of the platform itself and its large variety of applications needs to be done in automation, as it goes beyond the capabilities of human administrators. Also, this automation provides the opportunity to increase resource utilization and reduce costs [19]. Therefore, automatic placement and scheduling of data analytics workloads is an important topic in data analytics platforms.

2.2 Blockchain Fundamentals

A blockchain enables distributed ledgers that store data immutably in a secure and encrypted way. This section focuses on the fundamentals of blockchains technology, which we apply for improving data retention in data analytics platforms.

2.2.1 Blockchain Network and Consensus Algorithms

A blockchain, also known as shared ledger, is an append-only list of records, which are linked and secured using cryptography. Prominent blockchain implementations include Bitcoin [45], Ethereum [46] and Hyperledger [47]. Writes to the blockchain are called transactions. They are broadcasted to all nodes of the blockchain network. Moreover, multiple transactions are grouped and permanently stored into a single block. These blocks are in a chronological and linear way connected to form a blockchain, similar to a linked-list. Each block within the blockchain is uniquely identified by a hash of the block header. A block header includes the Merkle Tree [48] root that is generated by recursively hashing pairs of all transactions stored in a block until there is only one hash left. Additionally, each block stores the hash of a previous block header as a reference. Therefore, the sequence of hashes linking to its previous block creates a chain going back all the way to the first block ever created. Figure 2.7 illustrates the structure of a blockchain.

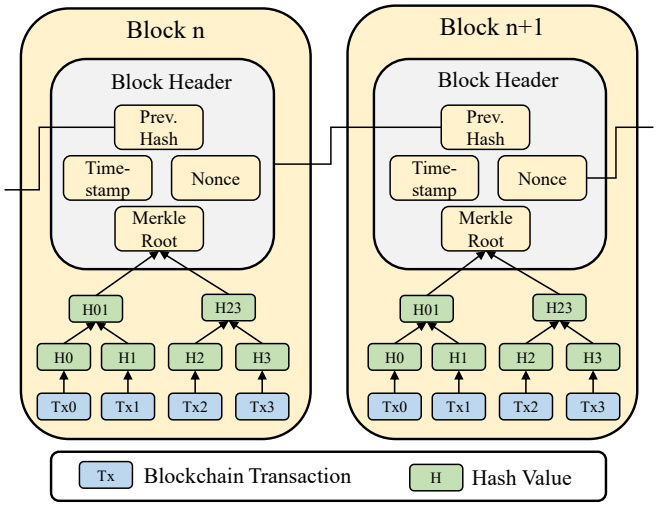


Figure 2.7: Data structure of a blockchain storing multiple transactions.

A blockchain is managed by a private or public peer-to-peer network that collectively validates and generates new blocks. Therefore, the blockchain network consists of multiple nodes, where each node has a local copy of the blockchain. Some nodes participate in a leader election process that determines which node gets the privilege to append the next block to the chain. These nodes, which are actively competing to become the leader of the next round, are called miners.

Most blockchains such as Bitcoin and Ethereum use Proof-of-Work as consensus protocol. At the start of each leader election round, all miners start working on a new computational problem, e.g. producing a hash, that depends on three different input data: new blocks of transactions, the last block on the blockchain and a random number. This refers collectively as block header for the current block. Each time a miner performs the hash function on the block header with a new random number, they get a new result. To win the election process, a miner must find a hash that begins with a certain number of zeros. Just how many numbers of zeros are required is a shifting parameter determined by how many miners and how much computing power is attached to the network. The first miner that solves the problem gets the privilege to write the new block with pending transactions that are not yet included in any block.

Motivation for participating and winning the election is a monetary reward. The winners may issue themselves an amount of the mined currency and they get to collect all fees that are charged for transactions. In order to have their transaction prioritized, users may offer to pay a higher fee. As a result, the blockchain forms a system that enables decentralized consensus without the need of any central authority. Beyond that, other consensus algorithms such as Proof-of-Stake exist, which select the generator of the next block based on combinations of random chosen values, instead of solving a computationally intensive problem.

2.2.2 Smart Contract-Based Blockchain

Smart contracts are programs to verify or enforce the negotiation or performance of a contract. They are automatically enforced by the consensus mechanism of the blockchain without relying on a trusted authority. The consensus protocol of the blockchain has the goal to ensure the correct execution of smart contracts.

Ethereum is a prominent smart contract-based blockchain. Its underlying Ethereum Virtual Machine (EVM) has a notion of global state including accounts, balances and storage. Each smart contract is stored on the underlying blockchain in bytecode, so called EVM opcodes with fixed definitions, and is executed by the EVM. Furthermore, any transaction must ensure a valid transition from the canonical state preceding a transaction to the new state it leaves the EVM in. Therefore, the order in which transactions are processed and written into a mined block is crucial. All EVM transitions are executed by every participant of the network and stores new states on the blockchain. The smart contracts inherit the blockchain properties of decentralization, zero downtime and security against fraud.

Smart contracts are written in Solidity, a Turing-complete bytecode language, which compiles into a special assembly code that can be interpreted by the EVM. Therefore, a contract is a set of functions, each one defined by a sequence of bytecode instructions. Figure 2.8 illustrates smart contracts through an example written in Solidity. It can receive money from other blockchain users, and its owner can split and send the collected money to others users, whose wallet addresses are stored in the outflow hashtable. The collected money is recorded via the balance variable, which cannot be altered by the program logic.

```
contract AWallet{
     address owner;
     mapping (address => uint) public outflow;

function AWallet(){ owner = msg.sender; }

function pay(uint amount, address recipient) returns (bool){
     if (msg.sender != owner || msg.value != 0) throw;
     if (amount > this.balance) return false;
     outflow[recipient] += amount;
     if (!recipient.send(amount)) throw;
     return true;
     }
}
```

Figure 2.8: A sample smart contract code [49].

Chapter 3: Related Work

Contents

This section presents related work on data and container placement strategies for data analytics platforms. First, related work on scalable data analytics systems that are widely used in data analytics platforms is presented. Second, related work on data, task and container placement strategies for these systems are presented. Finally, related blockchain-based data retention approaches are discussed.

3.1 Scalable Data Analytics Systems

In the last few years, data analytics platforms build up on shared-nothing commodity machines have gained a lot of momentum. They typically consist of a colocated resource management system and distributed file system, and multiple distributed processing system running on top in parallel. This section describes similarities and differences between systems of each class.

3.1.1 Distributed Dataflow Systems

This section describes related work on distributed dataflow systems, which we used for evaluating our placement methods. Distributed dataflow systems allow users to develop distributed applications. In general, they provide developers predefined operators known from functional programming such as Map and Reduce. They extend these operators with sequential code and connect them to form a dataflow graph, where edges represent the dataflow. After deploying a distributed dataflow system, users can submit these dataflow graphs as applications. The framework then automatically parallelizes and distributes the application on multiple nodes.

MapReduce [24] introduced a programming and an execution model for distributed data analytics on shared-nothing clusters. The programming model is based on the two higher order functions Map and Reduce. Both functions are enriched with UDFs. The execution models comprises the data-parallel execution of task instances of these two operations, where each Map phase is followed by a Reduce phase. In between both phases, the intermediate results are written to disk and shuffled. Moreover, the results of a Map task are sorted by a key, so that a successor Reduce task can read and reduce defined groups of elements efficiency. Fault tolerance is given because a distributed file system is used for storing intermediate results in between stages of map and reduce tasks. A major implementation of the model is the open-source Hadoop MapReduce [6]. Other work extend this implementation with iterations [50, 51] or SQL-like declarative language [52].

Nephele [53] and Dryad [54] added the possibility to express data analytics jobs in arbitrary Directed Acyclic Graphs (DAGs), instead of combinations of subsequent Map and Reduce tasks. Further work on Nephele under the name Stratosphere [55] extend it with different features. PACTs introduces second order functions to perform concurrent computations on datasets in parallel [56]. Meteor introduces an operator-oriented query language [57]. Hueske et. al [58] and Rheinlander et. al. [59] introduce various dataflow optimization techniques to Stratosphere. Ewen et. al [26] add support for bulk and incremental iterations, which are often used in graph and machine learning algorithms. Thamsen et. al. [60] use synchronization barriers between iterations to adapt resource allocation at runtime.

Spark [9] is another widely-used distributed dataflow system that offers the users a set of second order functions. The system relies on the concept of Resilient Dis-

tributed Datasetss (RDDs) to efficiently execute iterative and interactive jobs [61]. Moreover, Spark's RDDs provide fault tolerance using lineage. Compared to check pointing that writes intermediate results to disk, this can significantly speed up the dataflow computation. Furthermore, Spark provides higher-level programming abstractions including processing relational data with automatic query plans optimization [62, 63] and graph processing [64, 65]. Spark also provides stream processing features by using micro-batches [66].

Flink [10], the successor of the Stratosphere Platform, and Google's Dataflow [67] add further features regarding scalable stream processing. They are based on the concept of windows over continuous data streams. Furthermore, they provide mechanisms to cover late elements, which arrive after the system's event time clock has already passed the time of the late element's timestamp. Flink uses the stream processing engine for both, batch and stream processing.

3.1.2 Resource Management Systems

Resource management systems are orchestration software that automatically manage different applications and machines. They allow to share cluster resources among multiple users, applications, and frameworks by temporarily assigning resources to them through virtual containers. The rest of this section presents related work on resource management systems that support data analytics batch jobs.

Kubernetes [34], Firmament [27] and Borg [35] follow a monolithic scheduling design, where a central resource manager assigns containers and tasks to machines. Moreover, all upcoming applications are handled by the same scheduler logic. This uniform approach has lead to sophisticated schedulers. For instance, Paragon [36], Quasar [20] and Thamsen et. al. [29] use machine learning techniques to avoid negative interference between applications. Other examples are Maui [68] and the LSF platform [69] that involve different weight factors to determine placements and to support different policies. Firmament [27] shows that centralized scheduling approaches based on sophisticated algorithms can be fast enough to scale up to over ten thousands of applications and machines using a Google cluster workload trace.

YARN [11], Mesos [12], Nomad [70] and KOALA-F [71] follow a two-level scheduling design by separating the concerns of resource allocation and container

placement. In particular, different framework specific schedulers interact with a central resource manager that assigns partitions of the cluster resources for each application. Beside resource sharing, this allows to design more flexible container placement logic towards framework requirements. However, the framework-level schedulers do not have a global view on all available resources anymore.

In Mesos [12], a central resource manager offers resources to individual application-level schedulers, which can autonomously decide to accept or reject these offers. By accepting an offer, the framework schedule tasks on allocated containers. One advantage is that each framework can optimize the placement for their own goals. This allows to execute heterogeneous workloads that for instance cover web serving, batch analytics, stream analytics and machine learning more efficiently.

YARN [11] is part of Hadoop and supports a broad range of distributed processing frameworks such as MapReduce, Spark, Flink as well as machine learning frameworks like TensorFlow [72]. Users allocate resources for their application by specifying the number of containers and their size in terms of memory and CPU cores. In comparison to Mesos, resource allocations are request-driven. Framework specific schedulers request the central resource management for resources and receive container allocations in return. We used YARN for implementing and evaluating our placement strategies.

Omega [37] and Apollo [13] follow a shared-state design, in which the cluster state is independently updated by the application-level schedulers. The shared cluster is materalized in a single location, which is called 'cell state' in Omega and 'resource monitor' in Apollo. Moreover, both are coordinated using optimistic concurrency control. In Apollo, the shared-state is read-only and the transactions are directly sent to all nodes. Afterwards, each node checks independently for conflicts and accepts or rejects changes. By this, it keeps running even when the shared-state is temporarily not available.

3.1.3 Distributed File Systems

Distributed file systems handle datasets by splitting them into series of datablocks. These datablocks are placed among a set of nodes in a cluster. Moreover, each block is replicated and redundantly stored on multiple nodes. Thus, if a node stops

working or datablocks got corrupted unexpectedly, copies of the datablocks are available on other nodes for recovery.

HDFS [7] and GFS [73] are distributed file systems that focus on storing large datasets on commodity hardware for distributed data processing. GFS is a proprietary storage system developed by Google. HDFS is an open source implementation of GFS and the official storage system of Hadoop. Both systems focus on storing large files and high throughput for sequential reads and writes. Also, they follow a master slave paradigm, where a single master node is responsible for metadata management that includes maintaining the directory tree of all files in the file system and tracking where across the available nodes the file's datablocks are kept. The file itself is stored on the slave nodes, which serve as a pure data storage. The default datablock size in HDFS is 128 MB and in GFS 64 MB. Both systems focus on storing and providing fast access to large datasets for dataflow systems, which try to schedule their tasks directly on datablocks of the input file. By this, the task function can be executed locally on the node on which the input data is stored without additional network overhead. Storing all metadata in-memory on a central master has been reported as the limit for scaling out the number of files and datablocks in both systems. To overcomes this limitation, HopsFS integrates HDFS' central metadata service into a distributed in-memory NewSQL database [74]. Other authors introduce new file formats that merge multiple files into one that require a further indexing mechanism [75–78].

Ceph [8] is a distributed storage system whose base is an object-storage. It stripes datablocks across multiple nodes to achieve higher throughput, similar to a Redundant Array of Independent Disks (RAID)-0 that stripes partitions across multiple hard drives. Moreover, it replicates the datablocks on multiple nodes to provide fault-tolerance. In comparison to HDFS, it does not focus on accessing large files and thus, stripes datablocks with a smaller size of 64 KB per default instead of 128 MB. By this, it balances the load more effectively across the nodes and prevents bottlenecks in storage accesses.

GlusterFS [79] provides a POSIX-compliant distributed storage system. All nodes export a local file system as a volume. The GlusterFS client creates composite virtual volumes from multiple data nodes using stackable translators. In comparison to the other related storage system, it stores files without striping, and distributes and locates them using a hashing algorithm, instead of a using a metadata server.

Alluxio, formerly known as Tachyon [44], is an in-memory distributed file system. It can run on top of HDFS, GlusterFS or Amazon S3 [80] and uses the chosen system as a persistence layer. Alluxio itself is a memory management layer that buffers data in-memory and accelerates computation. Therefore, it provides high read and write performance for local data access. Moreover, it relies on the concept of lineage for fault-tolerance, instead of replication, to recompute lost data.

Furthermore, distributed dataflow systems support other storage systems to access data. Flink, for instance, provides connectors [81] to distributed databases such as MongoDB [82] and the proprietary cloud storage systems Amazon S3 [80] and Google Cloud Storage [83]. However, these connectors allow data access only remotely and therefore, achieving data locality is not possible, which can have a significant impact on the runtime performance of dataflow applications.

3.2 Placement Strategies in Data Analytics Platforms

This section describes different datablock, container, and meta data placement strategies. Scheduling is an important topic in the previously analyzed systems, because it directly affects the costs of operating a cluster. A placement resulting in a low utilization leaves expensive machines idle. A high utilization, on the contrary, can lead to antagonistic workloads and application combination that interfere with each other and decrease the performance of the running application [19]

3.2.1 Datablock Placement

A datablock placement policy determines the particular nodes for each datablock. Beside high scalability and fault tolerance, replication and file splitting can increase the performance of distributed dataflow engines, because tasks can access and process parts of the data in parallel stored on different nodes. However, when the processing task is not directly scheduled on the node where the input datablock relies on, the block has to be transfered through the network to the node executing the processing task. Especially in large data analytics clusters that comprise of hundreds or thousands of nodes, storing a large number of different files and running many different jobs at the same time, the input data and execution can be very likely

distributed on different nodes. In addition, data-intensive jobs that, for instance, join or merge two or more files require a lot of network resources, because the related datablocks are likely not to be stored on the same set of nodes. Therefore, efficient data placement is important in order to minimize the communication costs of data-intensive jobs.

Recent activities in data placement in data analytics clusters can be categorized in proactive data placement and active data placement.

The objective in proactive data placement is to place datablocks on desired nodes when they are loaded into the file system, and afterwards schedule job execution on these nodes. Examples are recurring jobs, in which data is loaded from another system into the data analytics clusters and afterwards, a data-analytic job is triggered on the new dataset. Examples can be found in click stream log analysis [84] or Surveillance Video Processing [85].

Coral [84] introduces a data and compute placement framework that jointly optimizes the location of data and tasks. It places input data and later job tasks on a small number of racks to reduce the load on the often oversubscribed core network and to improve the data locality.

CoHadoop [86], Hadoop++ [87] and GridBatch [88] enable the colocation of related files and its datablocks on the same set of data nodes based on user-defined property. Therefore, the user has to tag which files are related and should be colocated. In CoHadoop, the first file and its datablocks are distributed with the default data placement scheduling approach. For the second, CoHadoop places the datablocks on the same set of nodes like the previous file. CoHadoop focuses on technical issues and leaves the responsibility of choosing the placement of related files to the user and does not take data locality into account.

Amoeba [89] proposes a datablock placement technique for HDFS called hyper-partitioning. It generates many small partitions of data potentially from a different subset of attributes to answer queries by reading only a subset of partitions. AdaptDB [90] is a datablock manager for Amoeba that re-partitions the placement based on partitioning trees at runtime. In comparison to CoHadoop and Hadoop++ it requires no prior knowledge of the running jobs. Furthermore, the system introduces a join mechanism for Spark that identifies datablocks of the joining files that overlap on the join attribute.

Golab et. al. [91] automate the data placement process by proposing graph partitioning algorithms for computing nearly optimal data placement strategies for a given job. The objective is to decide where to store the data and where to place the tasks to minimize data communication costs. In contrast to CoHadoop, the workload must be known in advance. Additionally, they do not take care of parallel execution of tasks, which is an important feature of scalable dataflow systems.

The objective in active data placement is to move and change the number of datablock replications while they are residing in the file system. Techniques like Scarlett [92], ERMS (Elastic Replica Management System) [93], DARE (Adaptive Data Replication) [94], and Bui et. al [95] use file system logs and application access patterns to increase and decrease data the replication factor at runtime in order to reduce job execution time. Replications are spread across the whole cluster to avoid hot spots and increase the change of data locality. These systems are off-line systems, and their replication factor for each file is based on past accesses for that file. One drawback of these techniques are the additional network communication and storage overhead for dynamic replication. Ciritoglu et. al. [96] present a Workload-aware Balanced Replica Deletion algorithm (WBRD) for HDFS that goes in the other direction and decreases the number of replicas to reduce unbalancing and avoid hot spots.

3.2.2 Task and Container Placement

Containers provide a virtual abstraction in which the tasks of distributed dataflow systems are executed. Therefore, containers narrow the locations of nodes on which task by the framework itself can be scheduled and can have a significant impact on runtime performance. In the following we present different strategies for task and container scheduling focusing on data analytics.

YARN [11] supports per default four different schedulers. The fair scheduler that assigns resources across multiple users such that all users get, on average, an equal share of resources over time. Ghodsi [30] introduced the concept of Dominant Resource fairness (DRF) to YARN's fair scheduler that allows allocating multiple resources to users with heterogeneous demands. It determines each user's dominant resource and use it as a measure of the cluster usage and fair resource allocation. The capacity scheduler defines queues with resource quotas, for instance, giving

each user group a minimum capacity guarantee. The other two are First In, First Out (FIFO)-based and priority-based. Furthermore, Medea [97] introduces a YARN container scheduler focusing on long running applications such as streaming systems like Flink and Heron [98] and machine learning frameworks such as TensorFlow. All these YARN schedulers focus on resource allocation only, as it follows a two-level scheduling approach. The container placement is done on framework-level, so each framework implements its own placement logic by requesting containers from YARN's central resource manager.

Much research has been done on data locality, the idea of placing task or containers on nodes storing parts of the input data. For instance, in combination with fairness [38,39,92,99,100], in heterogeneous environments [101–103], or in virtualized environments by taking interference into account [104,105]. Most data-analytic frameworks such as MapReduce, Flink and Spark implement data locality as well.

Bell [106], SMiPE [107] and Ernest [108] use runtime estimation techniques to automatically allocate containers for distributed dataflow systems. Ellis [109] uses estimation techniques to dynamically scale distributed dataflows after synchronization barriers according to runtime targets. However, it allocates and releases containers without taking their node location into account.

Quasar [20], Bubble-flux [110], Heracles [111] and Thamsen et. al. [29] place containers by taking interference with colocated jobs into account. Quasar profiles unknown applications before they are executed. It does this by running sample runs on a few nodes. Afterwards, it is matched to previous jobs to classify the job based on collaborative filtering. Bubble-flux probes the nodes to measure the current pressure on the shared hardware resources. Moreover, it presents a method to predict how a running job will be affected by a potential colocated job. Heracles guarantees jobs resources due to coordinated management of multiple isolation mechanism. Thamsen et. al. [29] use a reinforcement learning algorithm to continuously learn good job container placements that are best executed simultaneously.

The proposed network-aware container placement method of this thesis combines data locality, interference and network distances as placement factors. It can adjust these factors differently based on user-defined weights. In comparison to two-level schedulers, our approach can adjust the placement logic per job, instead of having a specific scheduler per framework.

3.3 Blockchain-Based Data Retention

This section presents related work for using blockchain technology to increase data retention in shared storage systems, which are also used in data analytics platforms. First, related blockchains are introduced that are followed by data retention approaches using blockchain technology.

A blockchain, also known as shared ledger, is an append-only list of records, which are linked and secured using cryptography. Prominent blockchain implementations include Bitcoin [45], the first blockchain network implementation, and Ethereum [46]. Recent activities in blockchains can be categorized in bitcoin clones and alternative-chains.

Bitcoin clones copy the Bitcoin implementation and merely modify some of its parameters, for instance the reward, the consensus algorithm or the generation time of new blocks. Examples of Bitcoin clones are IXCoin [112] featuring a higher reward, BitcoinScrypt [113] using scrypt, a different proof-of-work algorithm that is resistant to GPU, FPGA, and ASIC implementations and Litecoin [114] using a block generation time of 2.5 minutes compared to the original 10 minutes.

Alternative-chains are not primarily designed as a currency. Instead, they represent a token to be used like a resource or a contract. Prominant examples are Ethereum [46] and Hyperledger [47]. Both allow developers to design Decentralized Applications (DApps) that run de-centralized on its peer-to-peer blockchain network and are not controlled by any single entity. Examples are real-life use cases, ranging from asset management to resource planning. Ethereum's DApps are based on smart contacts executed on its Turing-complete EVM. Hyperledger provides a framework to build business-oriented DApps.

Ghoshal and Paul [115] present an auditing scheme for cloud data that requires no third party involvement. Similar to our data retention approach, their approach allows to ensure and check integrity of a selected file based on blockchain technology. In particular, they propose to split the selected files into fixed-length file-blocks, hash the file-blocks, and generate a Merkle Tree per file based on all file-blocks hashes. The Merkl Tree root, the previous block hash, and other file metadata are then appended as a new block to the blockchain. In order to speed up the verification process, they use on a leaf number-based verification technique. The authors report

that their approach suits only well for files with rare updates, because changing a file leads to a modification of the corresponding block, and thus, any subsequent block needs to be rewritten. In comparison, we append the file metadata as a transaction to the blockchain and do not change any block due to transparency reasons. Additionally, their approach is not suitable for a public blockchain network, where modifications on already written and confirmed blocks are explicitly unintended.

ProvChain [116] is a data provenance system for cloud storage systems using blockchain technology. It monitors all operations on files and publishes them to a blockchain. ProvChain hashes data operations, constitutes a Merkl tree, and anchors the root node into a blockchain transaction. By this, it is possible to guarantee that data provenance was not tampered. ProvChain tracks files on operation level including read, move, and copy operations, and does not offer a functionality for a content-based file validation.

SmartProvenance [117] is a data provenance management framework for documents. Similar to our data retention method it is based on Ethereum's smart contract-based blockchain network. Furthermore, it uses the Open Provenance Model (OPM) to record the data trail immutable. In particular, it is based on access control policies and a voting mechanism to ensure that no malicious changes are made to the provenance data.

Blockstack [118] presents a blockchain-based naming and storage system. They present their knowledge about running a public key infrastructure service on top of Namecoin and how they migrate to bitcoin. In Blockstack, users can register and securely associate data with them. Only the owner of the particular private key can write or update the name-value pair.

Other systems like Filecoin [119], Permacoin [120], Storj [121], and SIA [122] provide decentralized storage services on top of a blockchain. Users can rent unused portions of their hard drive space in a peer-to-peer network. Metadata management and payment of the decentralized storage is done based on a blockchain network. Instead of rewarding miners for offering compute resources, they reward miners for offering their storage. ETH Drive [123] runs IPFS [124], a peer-to-peer distributed file system, and uses Ethereum as blockchain to provide tamper-proof data provenance to check data integrity.

Chapter 4: Problem and Concepts

Contents

This chapter describes the problem we address with this thesis including the state of the art and its limitations. Based on this, we introduce our approach and its methods to tackle this problem. The chapter concludes with assumptions and requirements for our methods and prototype system.

4.1 Problem and State of the Art

Runtime performance of data analytics applications is important to end users as they expect fast results and response times. In some cases, a poor runtime performance can even lead to negative financial consequences due to Service-Level Agreements (SLAs) violations. Runtime performance depends on many different factors besides increasing the amount of available resources per job. These include a precise control of application containers and input datablocks placements, job and algorithm specific

parameters, system configurations, dataset characteristics such as its partitioning and properties of the physical hardware as well as virtualization overhead.

Especially placement decisions can affect runtime performance, because, depending on the placement, more or less network traffic is required, which is often a major bottleneck in distributed data analytics. Furthermore, good placement decisions can avoid antagonistic workloads: application combination that interfere negatively with each other as well as many concurrent operations on the shared node's resources. Especially the latter can decrease the performance of applications, because the nodes resources can become overutilized and tasks executed on these nodes are likely to become a straggler. This, for instance, is the case when a node hosts multiple containers that require the same resource simultaneously, because containers are often executed without strict resource isolation in state of the art resource management systems. Another advantages is that a job's runtime performance can be optimized by good container and datablock placements without changing the application itself.

Data analytics workloads are diverse and include different types of batch and stream processing. This is because state of the art resource manager and storage systems are application-agnostic and thus, allow to run different frameworks and applications onto shared platforms. Furthermore, modern dataflow frameworks like Flink and Spark provide a unified engine for batch and stream processing including libraries for machine learning and graph processing. As a consequence, optimizing the runtime is difficult, as workloads are diverse. However, runtime performance optimizations based on container and datablock placement decisions are applicable to a broader range of jobs and require less application-specific and fine-grained configuration effort.

Furthermore, most state of the art systems such as HDFS and YARN leave some optimization opportunities out of account by placing all datablocks and containers with the same logic and without taking application-specific characteristics into consideration. For instance, they place datablocks and containers by spreading them on many different nodes, mainly due to provide high fault tolerance. However, when an application's containers is deployed in a randomly chosen and distributed cluster share, the chance of exploring data locality is limited by the particular nodes hosting the containers. This leads unnecessary network overhead and a worse runtime performance, because input datablocks may need to be accessed from a remote node, although the containers could have been placed on nodes that store parts of

the input data. And yet other applications, such as iterative programs, benefit from placing containers and tasks close to each other, which reduces the amount of data send through the network between tasks. Overall, selecting good node locations to optimize runtime performance is difficult, as its quality depends on the application type, available cluster resources and nodes.

In addition, a lack of support for long-term data retention exists. Besides the large amount of data that needs to be accessed and processed quickly, a lot of data exists that is less frequently accessed. However, this data is still valuable to organizations and thus, needs to be stored safely and reliably for a long period of time. For instance, eBay [125] stores hundreds of petabytes of data and analytics results long-term using a HDFS-based automated tiered archive storage. This results not only in more data volume that needs to be archived. Moreover, ensuring its integrity, validity, and provenance becomes an important, but also a difficult task, for instance, proving that a file stored a long time ago has not been altered without notice and, if it did, to track when it has been changed by whom.

4.2 Dynamic Data and Container Placement Approach

This section introduces the approach of this thesis to improve runtime performance of diverse data analytics workloads and to improve data retention by automatically selecting an appropriate placement strategy. First, an overview of the approach is given. Afterwards, the major components and methods are described in detail.

4.2.1 Solution Overview

A major goal of this thesis is to optimize runtime performance of data analytics applications in terms of its completion time. Therefore, its containers and input datablocks require precise control of their placement. However, different placement strategies can affect the runtime of a job significantly. For instance, depending on the application it can be beneficial to colocate or separate datablocks and containers across groups of nodes. Besides performance, long-term data retention is important to protect data, in case disaster or disruption occurs. Therefore, we argue that data analytics platforms should be improved with a dynamic resource and data

management system together with a higher degree of automation. This system should select an individual data and container placement strategy for each upcoming file and job, independently of the job's framework and type of application or file. It should select an appropriate strategy automatically based on data provided by the platform itself to decrease user intervention. Platform data includes job statistics of previous runs, available cluster utilization, datablock locations and dependencies between jobs and datasets.

Figure 4.1 gives an overview of our approach. The data analytics platform has two different user entry points, in which the datablock and container placement is triggered: One when a user or other application uploads or modifies a files. And another one, when a user or other application submits a job to run a data analysis task. Instead of executing a centralized placement algorithm, in our approach, a system for dynamic data and resource management receives this as input, before the actual datablock or container placement takes place. The green boxes present new placement methods and components designed in this thesis. Also, these are the major contributions of this thesis and will be discussed in the respective chapters in detail, as indicated in the figure. The blue boxes present monitoring and management components that we developed to monitor the underlying infrastructure and used as input to calculate appropriate placement decisions.

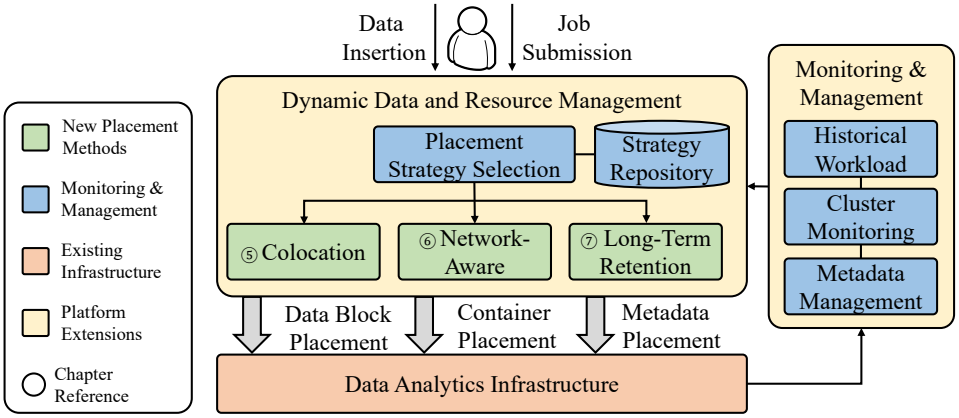


Figure 4.1: Dynamic datablock and container placement strategy selection to improve runtime performance and data retention in data analytics platforms.

The environment in which the approach operates is shown in Figure 4.2. Many nodes together form the processing and storage infrastructure. To be more precise, containers running on the worker nodes provide data analytics jobs a virtual environ-

ment for their tasks. Datasets are stored in a distributed file system, which splits the dataset into series of replicated datablocks across the nodes. Both are colocated to allow local data access. Furthermore, a monitor client collects resource utilization metrics per jobs on container level. All worker nodes are organized in racks with a hierarchical network topology. In addition, the infrastructure is integrated with a blockchain network, which provides an immutable data storage. We use it for storing metadata of what is happening in terms of changes to the datasets stored in the distributed file system.

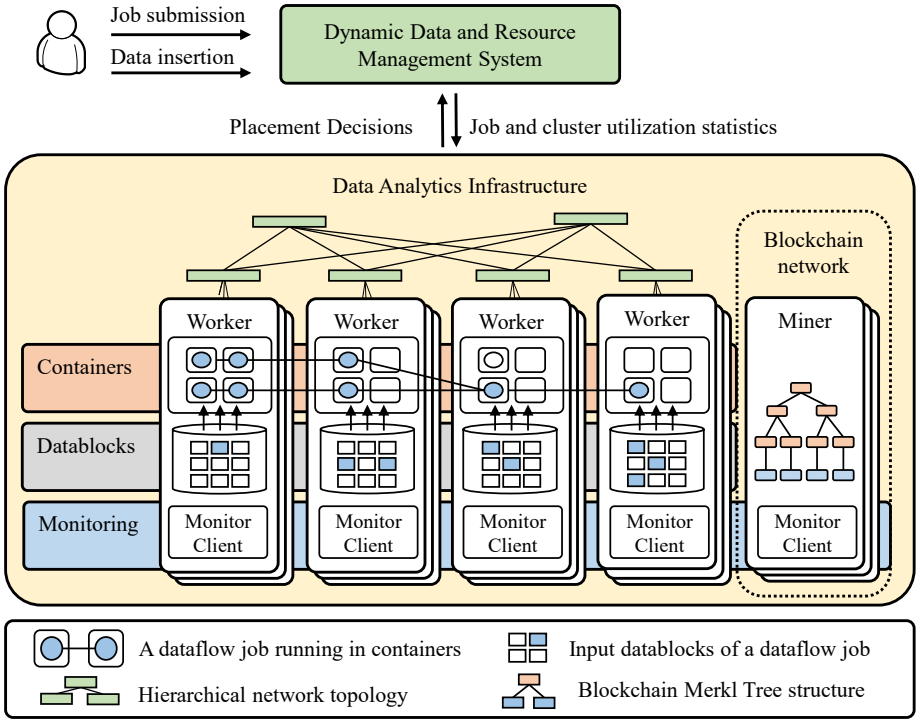


Figure 4.2: Overview of the environment in which the dynamic data and resource management approach operates.

4.2.2 Methods and Components Description

The proposed system consists of a *Strategy Selection* that shall automatically select a matching datablock or container placement strategy for each upcoming file or job from a repository of pre-defined strategies. The approach is shown in Figure 4.3. The focus of this thesis is to explore new placement strategies. Therefore, a matching between a both is currently done manually defined by a user. However, to reduce

user interventions and improve automation, the system is supposed to automatically select the strategy based on historical data and machine-learning techniques. As proof of concept, our approach consists of, but is not limited to, three different placement strategies to choose from: Colocation, Network-Aware and Long-Term Retention, which are described as follows.

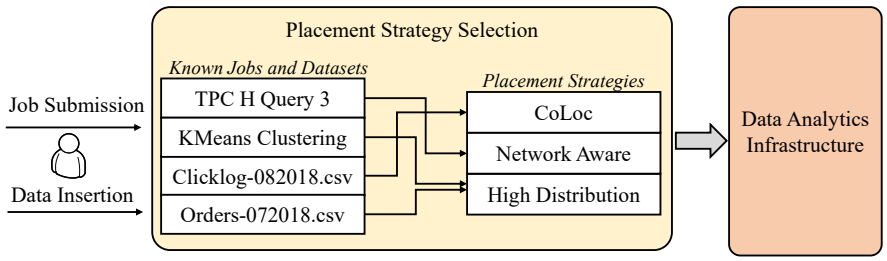


Figure 4.3: Overview of the placement strategy selection component.

Colocation is a datablock and container placement strategy that colocates both on a pre-selected set or subset of nodes. This strategy is mainly suitable for data-intensive recurring batch jobs. For this class of jobs it is possible to know in advance related files that are processed jointly by an upcoming job. In particular, the approach consists of a two-phase scheduler with a data placement and container placement phase. The data placement pro-actively determines on how many nodes and which particular nodes the corresponding datablocks of the file should be placed. Furthermore, it partitions the datablocks in a manner to reduce load imbalance and data skew. For this, it takes previous runs of that job, the current cluster utilization, dataset size, and other upcoming jobs into account. The container placement automatically determines the nodes storing most of the colocated input datablocks and places its container on these nodes. By this, distributed dataflow systems internally schedule tasks on local input splits of the data and reduce partitioning costs for several dataflow operations.

Network-Aware is a container placement strategy based on network distances between an application's input datablocks and their possible container locations. Furthermore, it takes into account that containers are executed with no strict resource isolation and thus, compete for the shared node resources such as CPU, disk, and network I/O. In particular, the strategy is based on a cost function that defines data locality, network distances and resource isolation in a weight cost function. Depending on the job, different weights can be beneficial. I/O-intensive jobs, for instance, tend to gain more performance advantage when their execution containers

are placed close to the input data, instead of placing them on a small group of nodes. In combination with job profiles and classification, the system can learn good weights automatically based on previous runs. In comparison to colocation, this strategy works for recurring and non-recurring jobs, where data is already stored without colocation.

Long-Term Data Retention is a strategy to improve the retention of stored data. Its approach is to detect and collect metadata of file changes and place them immutably on a blockchain network using dedicated smart contracts. By this, it is possible to prove that a file stored a long time ago in a shared storage system has not been changed without notification. Or, if it did, track when it was changed by whom. As the smart contracts are executed on the blockchain network, it is not possible to alter the stored data and the functionality of the data retention without access to the smart contract. Moreover, it explores blockchain network capabilities and operates autonomously without the need of a central controlling and trust entity. One method to automatically annotate files for data retention is through dedicated policies. For instance, based on their age and usage frequency, similar to policies used in automated tiered storages that assign data to a specific data tier. Putting both together provides a data retention approach with less user intervention needed.

Monitoring and Management features are required to gather various information for selecting and adjusting the most appropriate data and container placement strategy. Resource utilization metrics of executed jobs are recorded and stored in a central repository. These are required for job classification, for instance to determine if a job is I/O-intensive. Moreover, all file operations and job submissions are monitored. These are required for identifying files that are related and often processed jointly as well as for detecting changes that will affect data retention. Additionally, the current topology and available bandwidth is monitored. Also, we require the node locations of all datablocks, which we collect from the distributed file system. The component also stores user-defined information about recurring jobs and annotated files for data retention.

4.3 Assumptions and Requirements

Before describing the placement methods in detail, this section discusses assumptions and requirements to our approach. Shared data analytics clusters, distributed dataflow systems and batch processing workloads are assumed in this thesis. Observations leading to these three assumptions are introduced as follows.

4.3.1 Shared Data Analytics Clusters

We make two assumptions for the data analytics infrastructure. First, data analytics applications are executed on commodity clusters, in which compute resources are virtualized in containers and the input datasets are stored colocated in series of replicated datablocks. Second, these clusters are shared by multiple users and their applications and datasets. Thus, available resources are shared and users have more freedom to choose the most appropriate frameworks for their analysis task at hand. It is important to emphasize that for scheduling decisions, we assume full control over datablock placement and container placement. This is because both have large impact on the throughput and runtime performance of single applications as well as the entire platform.

4.3.2 Distributed Dataflow Systems

We assume workloads that consist of applications running on distributed dataflow systems. One reason for that is that they allow to execute a diverse mix of applications such as batch, stream, graph and iterative processing. We do not assume a specific distributed dataflow system. However, we assume that they are executed and integrated with a resource management system and a distributed file system. Moreover, we do not change the source code of the dataflow system itself. This is because our optimizations are based on the container and datablock placement level of the storage and resource management system to support a broad range of analytic frameworks and applications.

4.3.3 Batch Processing Workloads

We assume only batch analytic jobs as workload, which make a large portion in production clusters [37] and are executed in short-lived containers in the order of minutes [97]. Batch processing contains many different types of applications with various characteristics, such as ad-hoc queries, graph, iterative, and machine learning jobs. In this thesis, we do not assume long-lived allocations such as stream processing applications.

Moreover, we assume that some batch jobs are recurring jobs. In these jobs, the execution logic of a job stays the same for every execution, but the input data is changing for every run. Recurring jobs are for instance triggered when new data becomes available or at a discrete time for further analysis, e.g. hourly or nightly batch jobs. For these jobs, it is possible to adjust the selected placement strategy and its parameters to increase runtime performance in a more fine-grained way based on previous execution.

Studies of productive clusters show that a large number of jobs are recurring jobs [16, 17, 126, 127]. In numbers, Microsoft engineers reported that up to 60% of a 2700-node cluster are recurring batch jobs [17]. Another productive cluster study from Microsoft's Bing service shows that up to 40% are recurring [16].

Chapter 5: Data and Container Colocation Placement

Contents

This chapter presents CoLoc [128], a data and container placement method for optimizing runtime performance of batch applications in shared data analytics platforms. It focuses on recurring jobs, for which it is possible to know a priori which files are processed together. The method aims to place related datablocks and execution containers on the same group of nodes. As a consequence, it increases performance of some dataflow applications in comparison to using Hadoop's default placement strategies by a) reading more input data from local disks and b) having more local inter-process communication between containers, tasks and iterations.

The chapter is structured as follows. First, it describes the concept of CoLoc's related data and container colocation placement. Then it presents the system overview and placement workflow. Afterwards, the placement algorithms are discussed. Finally, it presents an evaluation based on different workload scenarios and applications from various domains using Flink as reference distributed dataflow system.

5.1 Colocating Related Data and Containers

We identified two possibilities to optimize job completion time without using more resources that are presented first in this section. The two-stage placement strategy that make use of these two optimization findings is introduced afterwards.

5.1.1 Optimization Goals

CoLoc's goal is to optimize runtime performance of data analytics batch jobs by improving the interaction between datablock and container placement. It does this by optimizing the usage of given cluster resources without allocating more compute resources per job. We identified two possibilities of improving runtime performance this way, by increasing data locality and by avoiding unnecessary data shuffling over the network.

The first optimization possibility is related to the decreasing chance of exploring data locality, due to the container virtualization introduced by the resource management system. When frameworks and jobs are deployed on such a system, its tasks are running in distributed containers that are colocated with the storage nodes of a distributed file system. Internally, these frameworks optimize local data access by scheduling its tasks on nodes storing partitions of the input data. However, the nodes, which host the containers for task processing, are determined before the framework is deployed by a resource manager. Thus, the internal scheduling possibilities of frameworks exploiting data locality are limited by the set of nodes chosen before. For instance, when a job gets a cluster share of eight nodes in a cluster with 128 nodes in total, the framework is not able to schedule its tasks on the 120 nodes outside of its allocated share, and thus, can't access a lot of datablocks locally. Therefore, runtime performance can be improved when containers are scheduled

directly on nodes storing datablocks of the job's input data, as the task schedulers of the frameworks can achieve a higher degree of data locality.

The second optimization possibility is related to the increasing data transfer between tasks over the network, when they run in containers highly distributed across different nodes. To be more precise, some dataflow operations can be very network-intensive. For instance, operators for group-based aggregations or joining two dataflows require all elements of the same group or with the identical join key to be available at the same task instance. Therefore, if the data is not already partitioned by these keys, the dataflow needs to be shuffled. In this case, all elements with the same key need to be moved to the same task instance, leading to all-to-all communication. The more the predecessor tasks are distributed across different nodes, the more data need to be shuffled across the network. Furthermore, iterative applications often tend to be network-intensive [129]. This is because the network is burdened by exchanging intermediate results between tasks and iterations, which are executed in data-parallel and require synchronization after each finished iteration. Placing tasks and containers of a job on the same set of nodes can reduce the network demand and optimize the job execution time. This is because inter-process transfer rates are higher compared to network exchange.

5.1.2 Two-Stage Data and Container Placement

We designed a two-stage placement strategy that makes use of these two optimization findings. The strategy consists of a data colocation phase and a container colocation placement phase that work closely together.

Data Colocation Stage

The strategy of data colocation is to actively place related files and their corresponding datablocks on the same set or subset of nodes, instead of highly distributing them among all available nodes. Furthermore, the selected nodes are prioritized for hosting job containers that process these datablocks. Besides improving the chance of local data access, data colocation enables to save network bandwidth by performing some parts of dataflow operations like joins locally. This is because involved datablocks are placed conjunctively and thus, such operations are partly

performed locally with less data-shuffling across the network. However, a container that executes the operation later needs to be placed on this node as well.

Figure 5.1 explains the method of data colocation by an example. It shows a cluster running a distributed file system consisting of five nodes. It stores multiple files, by which file A and file B are tagged as related. Both files consist of a replication factor of three. File A consists of two datablocks (A1, A2) and file B consists of three datablocks (B1, B2, B3). The main idea is to place the datablocks of both files on the same group of nodes. First, file A was loaded into the system and the strategy places their datablocks on the first and second node, except one block that is placed on a random chosen node to increase fault tolerance. Second, file B was loaded into the system. This time, it knows that file A and B are related, and partitions file B's datablocks evenly on the same set of data nodes. The algorithms to determine the node group size and its explicit nodes are described in Section 5.3.2.

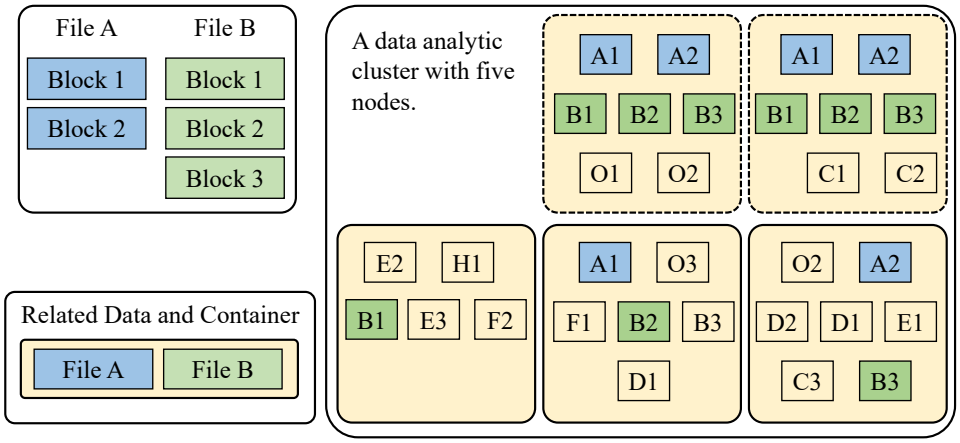


Figure 5.1: Illustrating the strategy of data colocation with two related files on a exemplary five nodes cluster.

Container Colocation Stage

The strategy of container colocation is to place a job's containers on the same set or subset of nodes storing the input data. Consequently, distributed data processing frameworks running in that containers are able to explore a higher degree of data locality, because they have direct disk access to the input datablocks. The nodes on which the containers are scheduled are determined and prioritized for job execution by the previous data colocation strategy. In addition, multiple containers of a job

are colocated on these nodes, so that intermediate results between stages are more likely to be exchanged via inter-process communication. One challenge at this point is not to place too many containers of the same job on the pre-selected group of nodes. This is because containers in resource management systems for data analytics such as YARN are executed with weak resource isolation. Furthermore, many containers have the same resource utilization profile, as its task inside execute the same data-parallel task. For instance, when performing a complex computation in a map function, CPU is likely the dominant resource. And when multiple of such containers simultaneously try to utilize the chosen nodes CPU or other resources, this can cause a decrease of performance. Therefore, it is important to maintain balance and not overload nodes with containers of the same job.

Figure 5.2 explains the method of container colocation by an example. It extends the related data colocation example of Figure 5.1 with a job A that uses file A and file B as input. The idea behind container colocation is to place the containers on the same set or subset of nodes, where the input data is residing in. Consequently, in the example the four containers (C1, C2, C3, C4) of Job A are distributed randomly on the first and second node. Internally, the nodes were reserved until a job with File A or File B was scheduled or no other cluster resource were available for cluster execution. In the case when the first and second node are busy, the scheduler waits a period of time or schedules the execution on other nodes. The phases' algorithms are described in detail in Section 5.3.2.

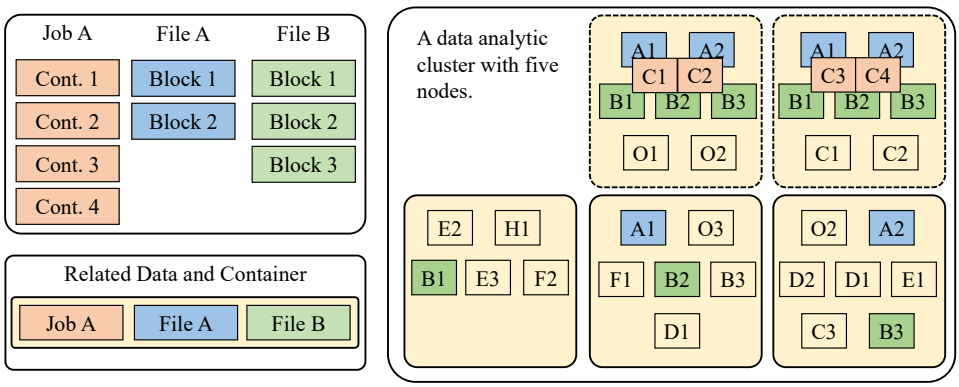


Figure 5.2: Illustrating the method of container colocation with a job accessing two related files on an exemplary five nodes cluster.

5.2 Placement Workflow and Components Overview

CoLoc is a placement system for data and container colocation enforcement. It is integrated with Hadoop's YARN and HDFS as well as Freamon [130]. Figure 5.3 gives an overview of CoLoc and its integration with the mentioned data analytics systems. The rest of this section describes CoLoc's components and its workflow.

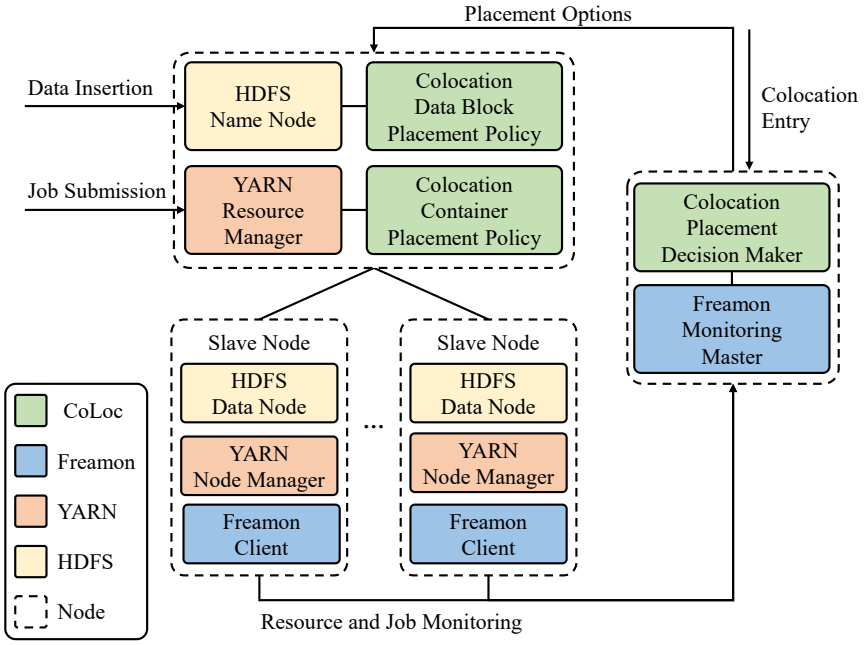


Figure 5.3: System overview and integration of CoLoc into Hadoop.

Name Node and Data Node are components of HDFS. The Name Node is responsible for metadata management and datablock placement. This includes maintaining the directory tree of all files in the file system and deciding and tracking where across the available nodes the file is kept. The data placement decision is made by a pluggable data placement policy, which we used to integrate CoLoc. The file itself is stored on the Data Node, which serves as a pure data storage.

Resource Manager and Node Manager are components of YARN. The Resource Manager is the central arbitrator of all compute resources and is responsible for scheduling jobs and containers on available resources. Similar to the data placement, container placement decisions are made by a placement policy, which we used to integrate CoLoc. A Node Manager running per-node is responsible for monitoring and managing containers. Inside the containers, the data analytics jobs are executed.

The Colocation Placement Decision Maker is the core component of CoLoc. It determines and provides scheduling options as guideline to the datablock and container placement policy. It is implemented in Java and consists of a REST API created in *Spark*[1]. Users and applications define via this API, which files are related and processed jointly by an upcoming job. The decision maker uses this information to determine on how many nodes and on which nodes related files and their datablocks should be placed based on the current cluster utilization and other files and job dependencies. Afterwards, it reserves the resources on these nodes for a matching job for a period of time and as long as the remaining cluster resources can cover the current workload.

Colocation Container Placement Policy is responsible for container placement. Similar to the datablock policy, it communicates with the Colocation Placement Decision Maker and checks before the container deployment and job execution starts, if related colocation entries exist. If this is the case, it places the containers on the provided colocated set of nodes. When no entry exists, the default container scheduling policy is used with prioritization on nodes that are not reserved for an upcoming job. Also, the component is implemented as a pluggable container scheduler, thus it can be used without any YARN source code modification.

Freamon[2] is the monitoring system that we developed to record resource utilization statistics for the containers of distributed applications. In addition, it provides live data about file changes, submitted applications, block changes, block locations as well as the current cluster utilization. The Colocation Placement Decision Maker uses the resource metrics to determine an appropriate set of nodes. It is implemented in Scala. *Akka*[3] is used for asynchronous message parsing of resource utilization metrics between the monitoring clients and their master. For collecting resource utilization statistics per node and container different libraries are used. The virtual file system `/proc` is used for recording CPU and memory utilization per container process. *Nethogs*[4] for recording the network utilization per container process. *PidStat*[5] for recording the disk and CPU utilization per container process. *Dstat*[6], to capture the overall resource utilization of a worker node. Moreover, Freamon uses

[1] `https://github.com/perwendel/spark` , accessed 2018-08-06.
[2] `https://github.com/citlab/freamon` , accessed 2018-08-06.
[3] `https://github.com/akka/akka` , accessed 2018-08-06.
[4] `https://github.com/raboof/nethogs` , accessed 2018-08-06.
[5] `https://github.com/sysstat/sysstat` , accessed 2018-08-06.
[6] `http://dag.wiee.rs/home-made/dstat` , accessed 2018-08-06.

Hadoop[7] REST API to get current cluster utilization metrics, application statuses, container locations, as well as data node information and datablock locations. All historical data is stored in *MonetDB*[8], a time series based database.

5.3 Related Data and Container Colocation Enforcement

This chapter introduces the data and container placement strategy in detail. First, parameters and definitions that are necessary for the designed placement process and its algorithms are introduced, which are presented afterwards.

5.3.1 Definitions and Parameters

The general purpose of the data colocation stage is to colocate datablocks of related files that serve as input for a recurring job. In a distributed file system, a file f is stored in series of datablocks db_i. Datablocks have a fixed datablock size dbs. Thus, the number of datablocks dbn per file is determined by $dbn = \lceil \frac{fs}{dbs} \rceil$, where fs is the total file size. Additionally, all datablocks are replicated with a datablock replication factor dbr across all available storage nodes. Both variables, dbs and dbr are user-defined per file. Formally, a file is defined as: $f : \sum_{i=1}^{dbn} \sum_{j=1}^{dbr} (db_{i,j},)$.

For recurring jobs, it is possible to know the input files and number of execution containers a priori. With this motivation, CoLoc allows users to define two different colocation entries c_i with the following characteristics. File Colocation $c_i : \{fl_i, ns\}$, where fl_i: $\{f_1, f_2, ...f_n\}$ is a list of related files and ns is the minimum number of nodes, i.e. node size, on which all datablocks of the c_i entry will be distributed. Folder Path Colocation $c_i : \{fp_i, ns\}$, where fp_i is a unique folder path containing files fl_i that are stored under the path. These files are defined as related without knowing in advance how many files will be stored under this path.

A job j describes a data analytics application. It consists of a unique job signature $jsig$, for instance the job's jar signature name. Additionally, a job has multiple user-defined configuration parameters. These parameters are often used as reference to the job's input file paths fi_i and output file paths fo_i. Other

[7] `https://hadoop.apache.org/docs/stable` , accessed 2018-08-06.
[8] `https://github.com/MonetDB/MonetDB` , accessed 2018-08-06.

parameters $jvar$ are often application and algorithm related such as thresholds, number of iterations, or keywords for filtering. Formally a job is defined as: $j : jsig, \sum_{i=1}^{fin}(fi_i,), \sum_{i=1}^{fon}(fo_i,), jvar$.

A job resource allocation jra describes the amount of resources a job gets for its execution. It consists of the number of containers cn and its resources per container in terms of memory $cmem$ and CPU cores $ccpu$. The relation between a specific job and a job resource allocation is given by the job signature $jsig$. A job resource allocation is specified by the user when submitting a job. Formally a resource allocation is defined as follows: $jra : jsig, cn, cmem, ccpu$.

A colocation description cd connects a job j and its input files fin with colocation nodes n_i, on which all related datablocks and containers of the colocation description entry will be placed with priority. The nodes size ns determines the maximum number of nodes. The colocation nodes $n_1..n_{ns}$ are determined by CoLoc based on the cluster utilization and existing colocation descriptions. An exp timer defines how long the nodes that are prioritized in the system, after all files are uploaded to the storage system. During that time, other jobs are scheduled on other nodes, if enough cluster resources are available. Formally a colocation description is defined as follows: $cd : \ j, \sum_{i=1}^{ns}(n_i,), exp$.

5.3.2 Placement Process and Algorithms

This section gives an overview of the algorithm and process, which is shown in Figure 5.4. First, before the data of a job is load into the system, a colocation description needs to be set. Afterwards, the nodes on which the data and containers should be colocated are determined and prioritized for this job. Next, the data and containers are placed with priorization on this group of nodes. Finally, after job execution, the prioritized nodes are released.

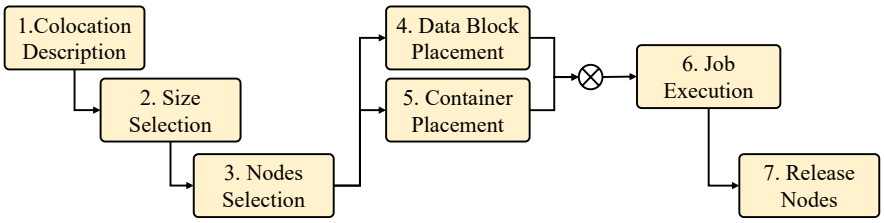


Figure 5.4: CoLocs Scheduling Process in a Process Diagram.

1. Colocation Description Definition

In order to use CoLoc, a colocation description must be specified by a user or another application. It contains the job signature *jsig* of the upcoming job, all its input files $\sum(fi_i)$, and an expiration timer *exp*. This description must be set before the datablock and container placement starts. Moreover, $\sum(fi_i)$ are placeholders for files that will be uploaded later into the file system. When all files are uploaded, the nodes are prioritized for related containers until the timer *exp* expires. CoLoc is primarily designed for recurring jobs, for which the input files and executing jobs can be known in advance. When no description entry exists, the default datablock and container scheduling takes place.

2. Determine the Size for a Job Description

The goal of this algorithm step is to determine a favorable node size *ns*, on which all upcoming datablocks and containers that are linked with a colocation entry *cd* should be placed. Algorithm 1 describes the procedure in more detail. The node size *ns* is automatically determined and adjusted, when one of the linked files of a colocation entry is loaded into the file system. Therefore, every time a new file nf is loaded into the system, all colocation description entries *cds* are selected based on its file name, where nf matches with one of the colocation description files $cds.fin$. Afterwards, for every matching *cd* two cases are traversed to determine a good initial node size *ns* for upcoming data and container placement.

1. The node size *ns* is set or adjusted to a higher value of an existing colocation description $cd.ns$. The $cd.ns$ parameter can either be set by a previous execution of step two or by the user manually. An adjustment can take place, because a file can be part of multiple *cd*. In this case, the value with the highest $cd.ns$ is selected.

2. Additionally, the *ns* parameter can be calculated by Freamon based on previous job runs. In this case, $Freamon.average()$ determines the average number of containers of all previous runs of job $cd.jsig$ that also used nf as an input parameter and whose total size of nf is within the threshold range of $nf.th$, which is important because the scale out behavior depends on the dataset size. Afterwards, the median number of containers *c* of these jobs are returned. In

order to calculate the node size *ns* for this case, we use $\frac{c}{nm.max}$, where *nm.max* is the maximum possible number of containers per node.

Algorithm 1 Node Size Selection Algorithm

```
1:  function ONSIZESELECTION(nf)
2:      ns ← 0
3:      cds ← ∀cd, where nf ∈ cd.fin          ▷ get all matching coloc descriptions
4:      for each cd ∈ cds do
5:          if cd.ns > ns then
6:              Adjust node size to a higher cd value
7:              ns ← cd.ns
8:          else
9:              Select node size based on historical average
10:             c ← 0
11:             c ← Freamon.average(cd.sig, nf.size, th)
12:             if c ≠ 0 then
13:                 ns ← c / nm.max          ▷ transform container size to node size
14:     return ns
```

3. Determine the Node Locations of a Job Description

In this algorithm step, specific node locations are determined, on which the data and containers of a colocation entry *cd* are placed. The procedure of the algorithm is shown in Algorithm 2. Similar to the previous step for determining the node size, this step is executed, when a new file nf is placed and a *cd* exists for that file. The approach is to first select all nodes *cd.nodes* that already store datablocks of a file that are related to nf. If these nodes are less than the determined node size, the favored node collection fn is filled up with not already used random nodes *rn*, until the size of fn equals to the requested size.

4. Datablock Placement

The purpose of the data colocation stage is to colocate datablocks of related files. The algorithm is shown in Algorithm 3. First, when a new file nf is loaded into the file system, the favorable node locations fn for that file are determined. This is done based on the previous algorithms. When any colocation description containing nf as input exists, all datablocks and its replicas of that files are distributed on the

Algorithm 2 Node Location Selection Algorithm

```
1:  function ONNODESELECTION(nf)
2:      cds ← ∀cd, where nf ∈ cd.fin        ▷ get all matching coloc descriptions
3:      fn ← ∅
4:      for each cd ∈ cds do
5:          Set or adjust node size
6:          cd.ns ← ONSIZESELECTION(nf)
7:          if cd.nodes ≠ ∅ then
8:              Set favorable nodes with known nodes
9:              fn.add(cd.nodes)
10:         while cd.ns > fn.size do
11:             Fill up favorable nodes with random nodes
12:             rn ← nodes.random
13:             if n ∉ fn then
14:                 fn.add(n)
    return fn
```

selected nodes fn. Thereby, for every upcoming datablock, the node is chosen that currently stores the minimum number of nf datablocks. Expect one replica for each datablock is stored on a random node, which is not in the favorable collection for improving fault tolerance. We distributed the datablocks on the select nodes uniformly to improve the chance to read datablocks in parallel. At the end of the algorithm, it is validated if all files of a colocation description are stored. If this is the case, the nodes are prioritized job container execution for $cd.exp$ time value.

Algorithm 3 Datablock Placement Algorithm

```
1:  function ONFILEPLACEMENT(curFile)
2:      fn ← NODESELECTION(nf)
3:      if fn ≠ ∅ then
4:          for each db_{i,j} ∈ nf do
5:              if j < rep then
6:                  n ← fn_{min db∈nf}
7:                  PLACE(db_{i,j}, fn.ran)
8:              else
9:                  r ← randomNode ∉ fnodes
10:                 PLACE(db_{i,j}, r)
11:         if all cd.fin exist then
12:             RESERVE(cd,cd.exp)
13:     else
          Use HDFS default block placement scheduler
```

5. Container Colocation Placement

The aim of the container colocation algorithm is to place a job's containers on the set or subset of nodes, where most blocks of its input data are already stored. The algorithm is shown in Algorithm 4. When a job is submitted to the cluster, CoLoc first parses the input file paths of the job submission $job.fin$. Afterwards, it is checked whether a colocation description for its job's input data exists based on the previous described Node Location Selection Algorithm. On success, the nodes are stored as favorable nodes fn. Afterwards, the container scheduler receive these nodes as preferences and places containers on these nodes. It is important to mention that the approach is best effort, so if not enough resources are available on these nodes, the remaining containers are distributed randomly across other available nodes with sufficient resources. Additionally, if the favorable nodes return an empty set, the containers will be placed with the default scheduling approach.

Algorithm 4 Container Placement Algorithm

```
1:  function ONCONTAINERPLACEMENT(job)
2:      fn ← NODESELECTION(job.fi)
3:      for each c ∈ job do
4:          if fn.size > 0 then
5:              n ← f.ran
6:              if n has enough resources then
7:                  PLACE(c, n)
8:              else
9:                  fn.rm(n)
10:         else
            Use YARN default scheduling behavior on non-reserved nodes
```

6. Job Execution and 7. Release Nodes

Finally, a job's container is deployed and the job execution starts. After the execution is finished and the containers are released, the colocation node reservations are released as well. By this, they are not related anymore and their datablocks locations can change again, for instance, when the data is not balanced across all nodes, because of adding or removing nodes from the cluster.

5.4 Evaluation

We evaluated CoLoc on a 64 node worker commodity cluster in a standalone and multi-job scenario. The evaluation is structured as follows: First, the cluster setup and experimental workloads are described in detail, which consists of standardized benchmarks and productive data analytics jobs. Afterwards, the results in terms of runtime performance of the workload scenarios are presented.

5.4.1 Cluster Setup

All experiments were done using a 64 worker node cluster. Each node is equipped with a quad-core Intel Xeon CPU E3-1230 V2 3.30GHz, 16 GB RAM, and three 1 TB disks with 7200RPM organized in a RAID-0. All nodes are connected through a single switch with a 1 Gigabit Ethernet connection. Each node runs Linux (kernel version 3.10.0), Java 1.8.0, Flink 0.10.1 and Hadoop 2.7.1.

One node acts as master and the other 64 as slaves. The master runs YARN's Resource Manager with CoLocs' Container Placement Policy and HDFS' Name Node with CoLoc's Datablock Placement Policy. This node also runs CoLoc's Placement Decision Maker for calculating and providing placement options to both components. All slaves are responsible for workload execution and each node runs YARN's Node Managers and HDFS's Data Nodes. In total, the cluster provides 896 GB memory and 256 cores, as we configured YARN's Node Manager to allocate 14 GB memory and 4 cores per node. We chose Flink as dataflow framework and use 3 GB memory and 2 task slots per Flink's Task Manager container as well as 2 GB per Application Master container, which runs Flink's Job Manager.

5.4.2 Jobs and Workload Description

The workloads of our evaluation consists of standardized benchmarks and productively used data analytics algorithms. The dataset size ranges from 8 GB to 250 GB per job. Furthermore, the jobs are diverse including relational database queries, machine-learning, and graph processing jobs. All jobs are implemented and executed in Flink. Table 5.1 gives an overview of the workload jobs.

Application	Type	Parameters	Dataset Size
TPC-H Query 3	Relational Database Query	-	250 GB
TPC-H Query 10	Relational Database Query	-	250 GB
K-Means Clustering	Machine Learning	8 clusters, 10 iterations	8 GB
Connected Components	Graph Processing	5 connectivity	25 GB

Table 5.1: Overview of the benchmark jobs used for evaluating CoLoc.

TPC-H [131] defines standardized benchmark queries for databases and transaction processing systems. We chose the TPC Benchmark suite H (TPC-H) Query 3 and 10. Both are business oriented ad-hoc analytical queries that examine large volumes of data. For each job, we generated a unique input dataset using the official TPC-H data generator.

K-Means Clustering [132] is a compute intensive data processing algorithm, which is used in the area of Machine Learning. It is an iterative algorithm that groups a large set of multi-dimensional data points into k distinct clusters without supervision. For our evaluation, we generated eight random fixed centers and 600 million points, resulting in approximately 8 GB input data.

Connected Components [133] is an iterative graph algorithm that identifies the maximum cardinality sets of vertices that can reach each other in an undirected graph. We used a label propagation-based implementation due to its better scalability and parallelization capability [134]. For our evaluation, we used a Twitter dataset with around 25 GB input data [135].

We designed a benchmark and performance evaluation tool to define, execute, and analyze cluster workloads called YARN Workload Runner (YWR)[9]. It allows to reproducibly measure and compare the performance and resource utilizations of CoLoc. The tool consists of two configuration files. A job configuration file allows to define different jobs and their parameters, and a schedule configuration

[9] `https://github.com/citlab/yarn-workload-runner`, , accessed 2018-08-06.

file allows a time-based schedule of these jobs based on their start time. At the end of an experiment, all results are stored in a central database for further analyzing.

5.4.3 Standalone Job Colocation Results

This section describes the results of evaluating CoLoc when execution a job in standalone. By this, all resources are exclusively available to a single job and no others are running on the cluster at the same time.

Figure 5.5 compares CoLoc with Hadoop's default datablock and container placement strategies. We report the execution times of a TPC-H Query 3 executed in 32 containers with a varying dataset size ranging from 125 GB to 1000 GB. The blue bar reports the job completion time when using Hadoop's default placement. The orange bar represents the job completion time when using CoLoc. The minimum number of nodes *ns* per colocation entry was set to eight. Thus, all data and containers of the colocated TPC-H job were placed on eight nodes in this experiment evaluation. Each of the experiments was done seven times. We report the median execution time. The results show that CoLoc can decrease job completion time between 19.51% and 31.19% depending on the input dataset size. Furthermore, the performance increases when the volume of processed data increases.

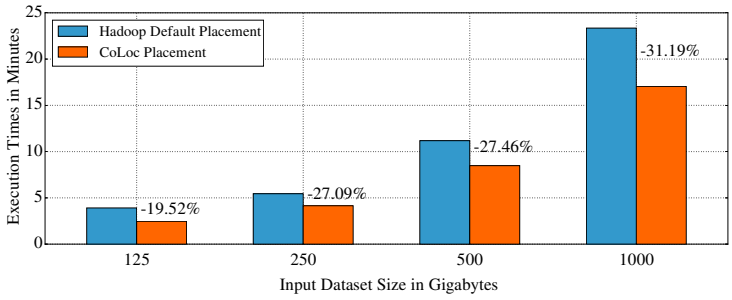
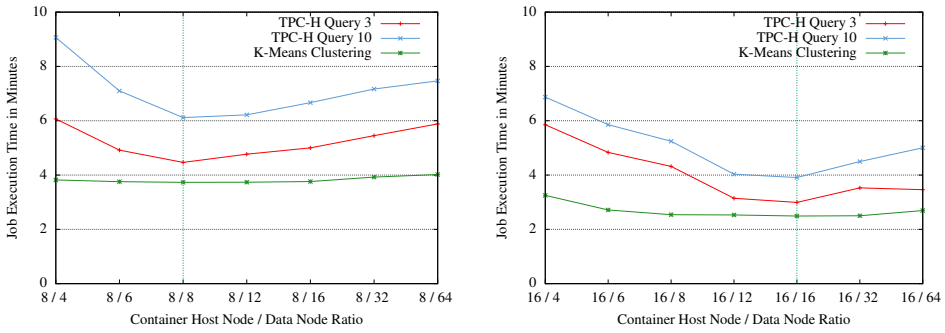


Figure 5.5: Comparing CoLoc with Hadoop's default placement strategy by running a colocated and non-colocated TPC-H Query 3 job with varying input sizes.

Figure 5.6 reports the effects on the execution time when using 32 containers and varying the cluster share of data nodes storing its input dataset. The number of data nodes increased for every iteration in the range between 4 and 64. Therefore, the data was partitioned differently for every experiment iteration, starting with a dense distribution on a few data nodes and ending with a high data distribution on

many data nodes. The x-axis defines the ratio between container host nodes and data nodes. Container host nodes are the number of nodes that host the containers. Figure 5.6a shows the results using 8 nodes as container hosts, whereas Figure 5.6b shows the results using 16 nodes as container hosts. The y-axis reports the execution time of a run, which was done seven times and the median time was reported.

The lowest execution times for all jobs are at a container host node and data nodes ratio close to 1, indicated with the dotted vertical line at $x = 8/8$ and $x = 16/16$. At this point, all containers are balanced evenly across the used data nodes and it can be reported that the data-intensive jobs TPC-H query 3 and query 10 benefit more from CoLoc than a CPU-intensive like K-Means. In numbers, using 8 nodes as container hosts, TPC-H query 3's runtime decreases 27.37% and query 10's runtime decreases 19.87%, whereas K-Means decreases only 7.40%. It is important to emphasize that in a placement without CoLoc, all datablocks are scheduled on 64 data nodes. Therefore, we use this 64 data node placement case as baseline value to compare the runtime variances when using CoLoc.

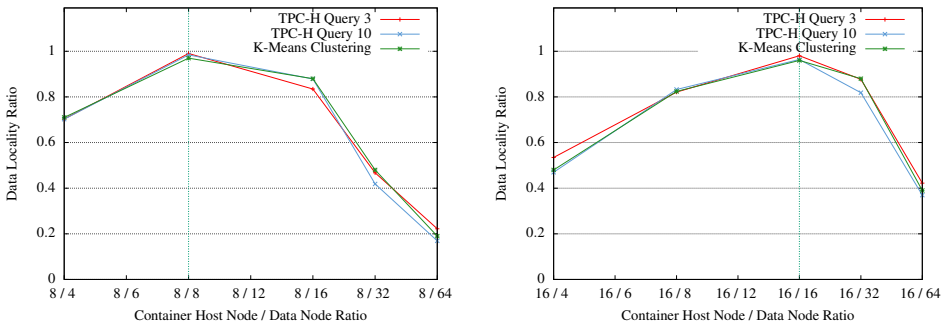


(a) Using 8 Container Host Nodes

(b) Using 16 Container Host Nodes

Figure 5.6: Execution times of various jobs using CoLoc with different Container Host Node and Data Node Ratios.

Figure 5.7 shows the degree of data locality for each run. When the container host node and data nodes ratio is close to 1, nearly all input data can be read from a local disk. A major reason for runtime variance is when the data is distributed on more nodes, as the chance of exploring data locality for the distributed dataflow framework decreases. This is because more datablocks are stored on nodes that do not host any of the job's container. Therefore, the framework internally cannot schedule its task on these nodes and needs to access it from remote disks.



(a) Using 8 Container Host Nodes (b) Using 16 Container Host Nodes

Figure 5.7: Data locality ratio of the TPC-H Query 3 and 10 and K-Means clustering on different CoLoc cluster shares.

5.4.4 Multi-Job Colocation Results

We evaluate the performance of CoLoc in a multi-job scenario, where multiple jobs share a data analytics platform. In particular, we execute eight jobs concurrently that allocate all available resources of the 64 node cluster. Furthermore, each job is accessing its own dataset stored in the shared distributed file system.

The workload of this experiment consists of two K-Means clustering (KM), two Connected Components (CC), two TPC-H Query 3 (T3), and two TPC-H Query 10 (T10) jobs. The job submission order was defined by: $\{T10_1, KM_1, CC_1, T3_1, T10_2, KM_2, CC_2, T3_2\}$. In addition, the number of nodes *ns* per colocation entry was set to eight. When the datasets were loaded into the system, CoLoc checked if the data was tagged with a colocation entry. If this was the case, the data was colocated and the nodes were prioritized for later processing. The described workload was executed seven times by using YWR. We report the median of all job runs.

Figure 5.8 reports the execution times of the first multi-job benchmark. It compares two workload situations. In the first scenario, the workload consists of three colocated jobs using CoLoc, and five non-colocated jobs are scheduled with the default Hadoop schedulers. We colocate three of the eight jobs, because in production clusters, it is reported that around 30 % are recurring and for these jobs, it is possible to colocate. The job's execution times of this workload are represented as orange bars. The three colocated jobs are represented as orange bars with black

vertical stripes. In the second scenario, the workload consists of the same jobs. However, this time all are placed the default scheduling behavior of Hadoop. The blue bars show the execution times of all jobs of this Hadoop based workload. We compare both workload scenarios with each other.

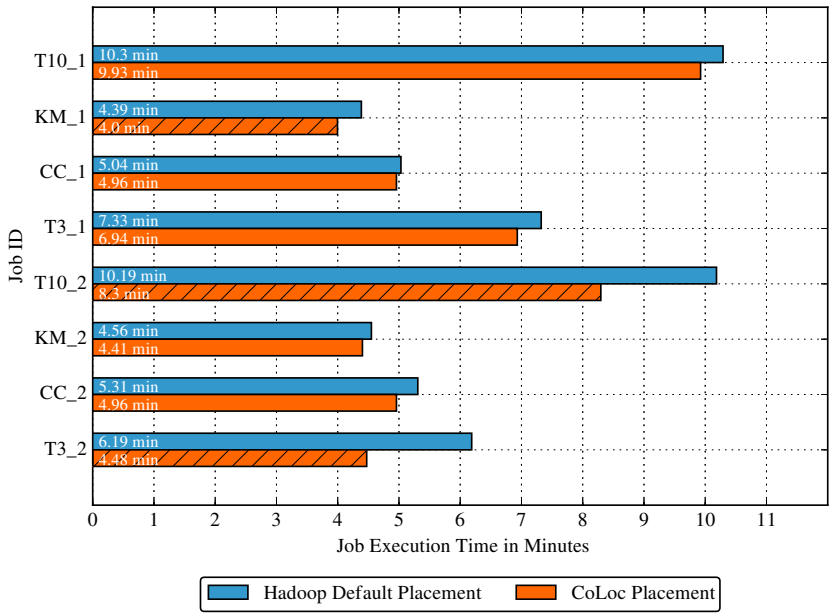


Figure 5.8: Job runtimes of a workload consisting of three colocated jobs and five non-colocated jobs.

In this benchmark scenario, CoLoc reduces the execution time for colocated jobs by average 20.63% ($KM_1 = 9.29\%$, $T10_2 = 20.43\%$, $T3_2 = 32.16\%$). The execution time of colocating jobs decreases, because less network congestion occurs and more data was read from local disks. Additionally, colocated jobs are running more resource isolated with less interference in its own cluster shares. In addition, we report that non-colocated jobs do not have any negative change on their execution time. In the experiment they even benefit by average 4.11% ($T10_1 = 3.63\%$, $CC_1 = 1.47\%$, $T3_1 = 5.48\%$, $KM2_2 = 3.28\%$, $CC_2 = 6.79\%$,) less job competition time from CoLoc. A reason for that is that their execution containers are out of necessity placed on a smaller group of nodes, which increases the inter-process communication between containers. Taking all jobs into account, CoLoc decreases total job completion time by 10.32%. Further benchmarks show that by colocating all jobs, it is possible reduce execution time by average 34.88%.

In Figure 5.9, we execute the same job sequence with a delay of 30 seconds after each job. We do this, because in the previous scenario, we had clear network-intensive phases. For instance, when all jobs are in the same stage like at the beginning, when they all start reading the data from the distributed file system. In this delayed scenario, CoLoc decreases the average execution time of colocated jobs by 13.69% ($KM_1 = 8.1\%$, $T10_2 = 13.96\%$, $T3_2 = 19.02\%$). We have less performance gain in comparison to previously workload scenario without any delay. This is because in this scenario less phases overlap and concurrent reads from local disks occur. Taking all jobs into account, CoLoc decreases total job completion time by 7.07% of this delay workload scenario. Hence, there is less performance gain. However, it is important to emphasize there is not a single job, regardless of whether colocated or not, that has a worse runtime.

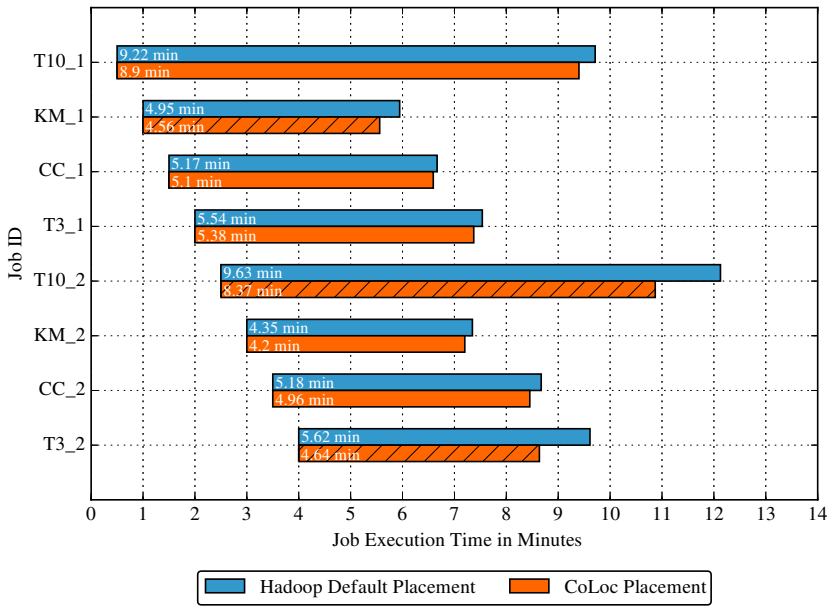


Figure 5.9: Job runtimes of a workload consisting of three colocated jobs and five non-colocated jobs scheduled with a 30 second delay.

Chapter 6: Network-Aware Container Placement

Contents

NeAwa [136] is a container placement method that aims to increase runtime performance of dataflow batch applications in shared data analytics platforms. It takes various network information for container placement into account. This includes distances between possible nodes hosting a job's execution containers as well as distances to and between nodes storing a job's input datablocks. Furthermore, the method does not assume recurring jobs like CoLoc and thus, it can be applied also on jobs that are executed only once or for the first time.

Finding a good container placement in large data-analytic platforms based on network distances is an optimization problem with a potentially huge search space. Therefore, we choose SA, a probabilistic method to find an approximation of the global optimum of a given cost function in a fixed amount of time. Our cost function reflects network distances between a job's containers and datablocks as well as other characteristics such as container isolation and datablock replication. NeAwa's placement strategy is integrated with Hadoop YARN and HDFS and we compare it with Hadoop's default placement strategy. For the evaluation, we used Flink and set-up a testbed with a hierarchal fat-tree topology and evaluate our method with different workloads.

This chapter is organized as follows. First, NeAwa's concept is introduced. After that, a system overview and integration with other data-analytic systems is given. This is followed by a description of the SA-based placement algorithm including its cost function. Finally, an evaluation of NeAwa is presented.

6.1 Placing Containers Network-Aware

The approach of NeAwa is a job's container placement based on network information and locations. Therefore, it is based on network distances between each other, and between containers and nodes storing the input datablocks. For this, it takes the current network topology, possible container locations, interference from other containers, job's input datablock location and current data analytics cluster utilization into account when scheduling containers. Many applications can benefit from this optimization that reduces network demand, resulting in lower completion times. However, we found out that there is a trade-off between placing a job's containers close to the input data and placing them close to each other, to which different jobs benefit differently.

6.1.1 Data-Locality versus Container Closeness

Placing a job's containers directly on nodes that store the input datablocks leads to a high distribution of containers on different nodes. The reason of this distribution is due to the datablock placement strategy of the underlying distributed file system

primary designed with focusing on fault tolerance. Datasets are stored in series of datablocks that are replicated and spread across different data nodes. Besides fault tolerance, this increases access performance allowing to read a file's datablocks in parallel from different disks and nodes. The node storing a particular datablock is determined by the distributed file system itself without taking its later processing location into consideration. As a consequence, datablocks are highly distributed and by placing the containers on nodes storing most of the input datablocks, the containers are highly distributed as well. The main advantage of this data and container colocation is that jobs achieve a high degree of data locality and access more data from local disks. However, communication between successor tasks is often carried out through the physical network.

Placing a job's containers close to each other, so that they run on a small group of nodes increases network throughput between these containers. This is because they are partly placed on the same physical host or on the same rack, in which the available bandwidth is higher than between nodes in different racks. Yet, some jobs benefit from placing containers and tasks close to each other on the same set of nodes. For instance, jobs with operators for group-based aggregations or joining two dataflows require all elements of the same group or with the identical join key to be available at the same task instance. Therefore, if the data is not already partitioned by these keys, the dataflow needs to be shuffled. The more the predecessor tasks are distributed across different nodes, the more data need to be shuffled across the network [25]. Another example are iterative jobs [26, 137]. For instance, Page Rank iteratively updates a rank for each node of an input graph by summing rank contributions of adjacent nodes. In each iteration a shuffle is required for aggregating values to compute page ranks, and a second shuffle is required to update each page rank. The more iterations are executed and its tasks are distributed among different nodes, the more data needs to be shipped over the network. However, on the contrary to the previously described idea of placing containers directly on nodes that store the input datablocks, placing containers a smaller set of nodes decreases the chance of achieving data locality. This is because reading data locally is limited to the set of nodes on which the containers are placed.

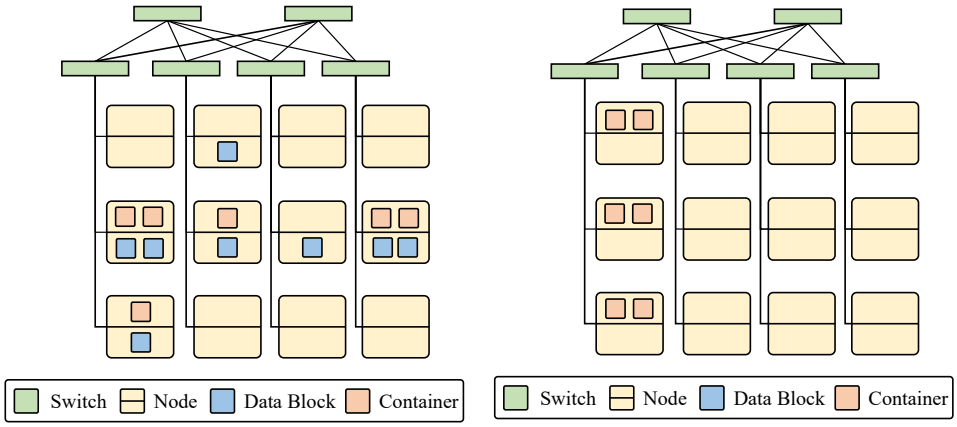
Furthermore, containers in data-analytics clusters are executed with no strict resource isolation, and thus, they compete for shared node resources such as CPU, disk and network I/O. The advantage of this method is based on the fluctuating resource demands of long-running analytics jobs. By running multiple jobs colocated in this

way, it is possible to increase resource utilization and overall throughput due to statistical multiplexing [32]. Since dataflow systems are based on data parallelism, its tasks and containers execute the same program logic in parallel and thus tend to stress the same resources at the same time. For this reason, resources of a node in the best case are shared by containers of different jobs. For NeAwa's idea of placing a job's containers on a small group of nodes, this implies that some resources of these node should be allocated by other or even no jobs, instead of allocating all resources for one particular job.

NeAwa focuses and optimizes placements in large and shared data-analytic clusters that are often organized in hierarchical networks like fat trees [138]. In such designs, nodes are grouped into racks of 20 - 80 nodes at the lowest level and multiple paths with different hop counts can exist between two nodes of different racks [39]. However, bandwidth between nodes within a rack is higher than the bandwidth between nodes in different racks. This is because the network and its switches are often oversubscribed and blocked for cost saving and maintenance reasons. A blocking network means that the number of links per switch that go to the upper level are lower than of those to the bottom level. Therefore, less switches are needed, however, paths between switches are shared by multiple nodes. Authors report productive data analytics clusters with a blocking factor of 1:5 [32, 40]. At the same time, local storage mediums become faster and popular in data analytics clusters such as SSDs [41–43] or virtual in-memory storages like Alluxio [44]. As a consequence, it is favorable to reduce network bandwidth in the core network.

6.1.2 Network-Aware Placement Strategy

The goal of archiving high data locality and placing containers close to each other on a small set of nodes acts antagonistically to each other. In Figure 6.1a, containers are colocated with the input data, and thus spread across a large number of nodes. On the contrary, in Figure 6.1b all containers are placed on a dense group of nodes with less network hops in between. Furthermore, it is important to emphasize that data analytics jobs have different characteristics. Data-intensive jobs benefit often more from achieving data locality, whereas some CPU-intensive and iterative jobs benefit more from being executed on the same set of nodes. Therefore, the NeAwa placement strategy is based on a dynamic weight cost function that takes both into account.



(a) Data locality, placing containers close to nodes that store the input datablocks.

(b) Container togetherness, placing containers close together on the same set of nodes.

Figure 6.1: Comparison of both container placement goals data locality and container closeness.

- **Data locality**. Placing containers close to the job's input data to allow more local reads from the underlying distributed file system as remote reads cause network traffic.

- **Container closeness**. Placing a job's containers close together with a low number of network hops in between them to reduce network traffic on links in the core network and by taking weak resource isolation into account.

Container closeness also reflects load balancing. This is because containers share the resources of the nodes they are placed on. Consequently, the container load should be balanced over the selected set of nodes.

Depending on the application type, we can weight both factors differently. Currently, we use dry runs with different weight factors and choose the best weight for an application. However, in combination with job profiles and classification, the system could learn good weights automatically based on previous runs.

Furthermore, in large deployments with hundreds or thousands of nodes, finding good container placements is an optimization problem with a potentially huge search space [139]. For this reason, we use SA, a probabilistic method to find an approximation of the global optimum of the given function in a fixed amount of time. Our SA-based algorithm and cost function is described in Section 6.3.

6.2 Placement Workflow and Components Overview

NeAwa is integrated with the resource management system YARN, the distributed file system HDFS, the dataflow engine Flink and the Software-Defined Networking (SDN) controller OpenDaylight. The architecture, which is shown in Figure 6.2, follows a master and slave model.

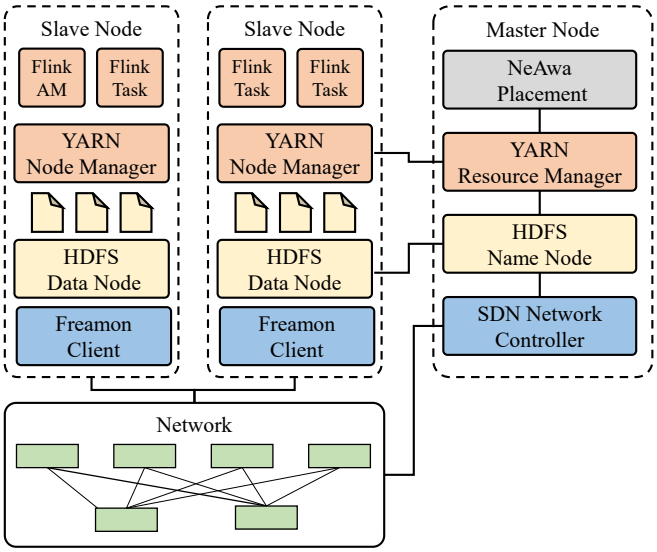


Figure 6.2: NeAwa's System Overview and Integration.

Node Manager and Data Node are the slave components of Hadoop's YARN and HDFS. Both are running on each slave node, which makes up the majority of nodes, and are responsible for storing the data and running the data-analytics computations within containers. In addition, a Freamon Client runs on every slave node to collect container utilization metrics that can be used to generate detailed job profiles. These profiles can be used to automatically adapt NeAwa's placement configuration parameters to gain knowledge of previous job runs.

Resource Manager, Name Node and Network Controller manage data storage, network and computing functionalities. The YARN's Resource Manager receives job submissions from the users, and is responsible for allocating needed resources and containers. Moreover, it provides cluster and node utilization statistics. The SDN [140] network controller automatically gets the current network topology and network utilization on the core network. The HDFS' Name Node provides information about datablock localities, which we use to improve data locality.

For later evaluation, we used Flink. Every job consists of one container running an Application Master (AM) that coordinates and monitors all n execution containers of a job, which are, for instance in Flink called Task Manager. The Application Master requests the Resource Manager for resources and receives container allocations in return that are confirmed by NeAwa.

NeAwa Placement is the core component and contains the logic for our placement algorithm. It decides where to place a job and its container best based on available resources, topology information, block location and running applications. The component uses the existing REST interface to all other master components.

Figure 6.3 shows a sequence diagram to illustrate the work flow between all system components to find a good container placement. This process is executed, when NeAwa was selected as container placement strategy.

First, an application is submitted to YARN's Resource Manager. The submission contains the amount of needed containers and their computation specification as well as the path of the input data.

The Resource Manager deploys an Application Master on the available slave node. The Application Master is responsible for allocating containers from the Resource Manager. Therefore, it first requests our placement component NeAwa where to place containers. NeAwa receives an application request specified with a resource profile, which contains the amount of needed containers and its virtual cores and memory demand per container as well as distributed file input path.

Afterwards, the NeAwa component calculates container placement hints based on information provided by the Resource Manager, Network Controller and Name Node including available resources, running application containers, distributed file system path and network topology.

The result are placement hints that are send back to Application Master, which afterwards sends container requests for these hints to the Resource Manager. Finally, the Resource Manager will allocate the containers and the execution of the application starts.

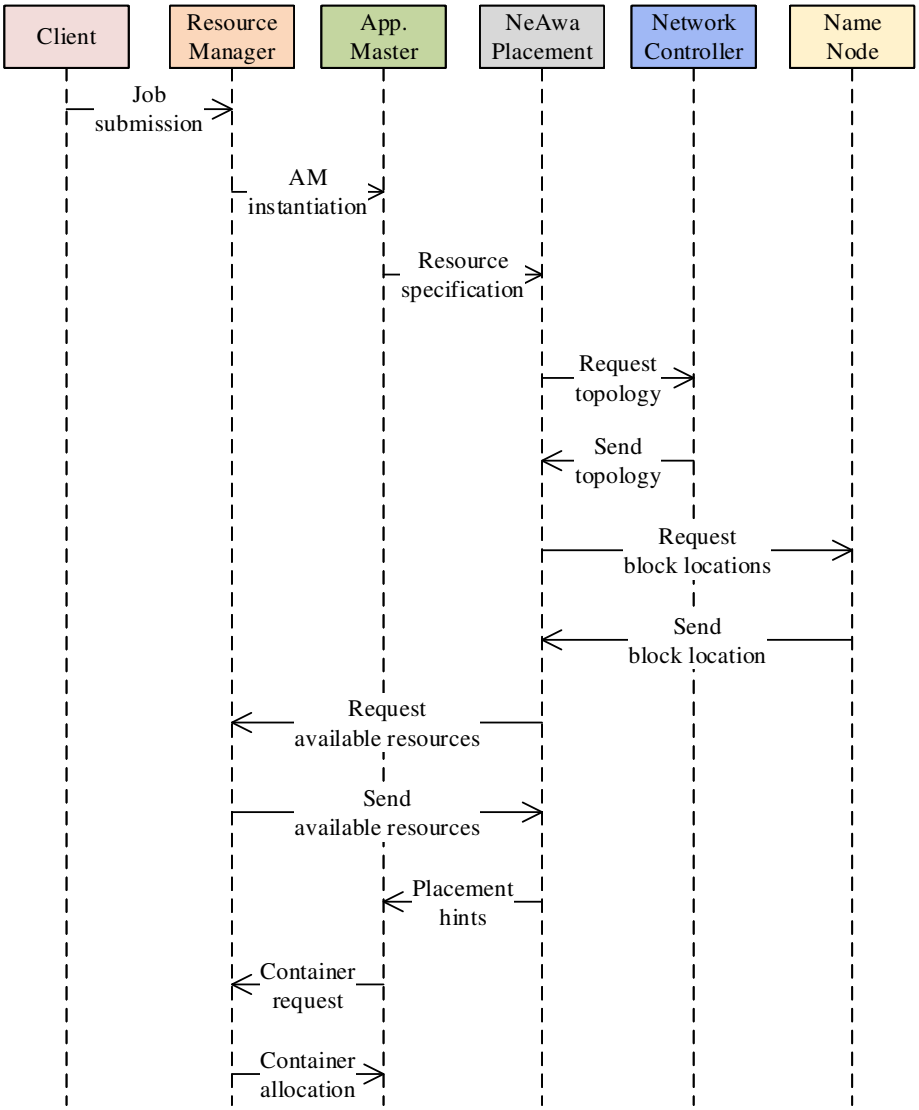


Figure 6.3: NeWa's components interaction and placement process.

6.3 Placement Method and Algorithm

The goal of our network-aware container placement is to find a good set of nodes for hosting a job's containers that provides good runtime performance. This is achieved by incorporating the objectives container closeness and data locality into the placement decision making process. Container closeness aims to place containers of jobs close together with a low number of network hops in between

them to reduce network traffic on links in the core network and by taking weak resource isolation into account. Data locality aims to place containers close to their application's input data to allow more local reads from the underlying distributed file system as remote reads cause network traffic. We use SA, a probabilistic method to find an approximation of the global optimum of the given function in a fixed amount of time, to find a good container placement. The cost function we propose consists of two main components, reflecting the objectives container closeness and data locality. All components are normalized and assigned with weights, depending on their importance and the cluster infrastructure.

6.3.1 Placement Algorithm Using Simulated Annealing

This section describes the network aware container placement algorithm in detail. Finding good placements for a job's containers in large data-analytics clusters is an optimization problem with a potentially huge search space. For this reason, the algorithm is based on SA [141], a probabilistic method to find an approximation of the global optimum of the given cost function. As cost function, we use and combine the previously defined container closeness and data locality terms. Moreover, both terms can be weighed manually or depending on previous runs of the job. The cost function itself is described in the next section in detail.

We select SA for determining a good container placement for two reasons. First, it allows to move from a initial random container placement selection to one with a low cost in a fixed amount of time. Second, during this finding process, it avoids to getting caught at local maxima, which are container placements with lower costs than other placements with just a few containers on different nodes, but which are not the optimal placement. In particular, our algorithm consists of five steps. Starting with a random container selection and iteratively finding a least-cost container placement. The following describes the algorithm steps in more detail.

1. **Initial Random Placement.** Generate an initial random container placement. Therefore, nodes with sufficient resources for hosting a job's containers are selected randomly.

2. **Cost Calculation.** Calculate the container placements cost using our network-aware cost function and weights.

3. **Random Neighboring Placement.** Generate a new container placement that is close to the previous placement. In particular, we switch the node of one randomly selected container of the previously placement with a new randomly selected node with sufficient resources.

4. **Cost Re-Calculation.** Re-calculate the cost of the new container placement.

5. **Placement Selection.** Compare the costs of both placements.

   - If $C_{newCP} \leq C_{oldCP}$: If the new placement has a smaller cost than the old placement, the new one is selected as the base for the next iteration. By this, the placement is iteratively getting closer to an optimum.
   - If $C_{newCP} > C_{oldCP}$: If the new placement has higher costs than the old placement, an acceptance probability decides to keep the worse placement or not. By this, it is possible to chose a worse solution and to get out of a local maxima.

6. **Final Placement.** Steps 3-5 are repeated until a maximum number of iterations is reached.

Algorithm 5 presents our container placement algorithm in detail. The *anneal* method is the entry point and is executed when a job is submitted and the network aware placement strategy is selected for its container placement.

As SA was inspired by a method of heating and cooling metals, the maximum number of iterations is determined by a temperature function. Therefore, the algorithm consists of a temperature parameter T. NeAwa starts with $T = 1$ and is decreased at the end of each iteration by multiplying it with a constant $\alpha = 0.9$ until it reaches $T_{min} = 0.00001$.

In each temperature iteration, a neighbor-cost-compare is done. Therefore, the previous container selection is slightly changed by choosing one new node with sufficient resources for one randomly selected container, which is shown in the *neighbor* function. In our algorithm, this comparison is done $i = 100$ times.

The cost comparison is based on an acceptance probability function ap. As input, it takes in the cost of the previously placement C_{old}, the current placement C_{new}, current temperature T, and the Euler's number e into account. It returns

Algorithm 5 Network aware container placement algorithm based on simulated annealing.

```
1:  function ANNEAL(cp)
2:      C_old ← COST(cp)
3:      T ← 1.0
4:      T_min ← 0.00001
5:      α ← 0.9
6:      i ← 0
7:      while T > T_min do
8:          while i < 100 do
9:              cp_new ← NEIGHBOR(cp)
10:             C_new ← COST(cp_new)
11:             ap ← e^((c_old − c_new)/T)
12:             if ap > RANDINT(0,1) then
13:                 cp_old ← cp_new
14:                 C_old ← C_new
15:         i++
16:     T ← T * α
        return cp_old
17: function NEIGHBOR(cp)
        //Get all possible container allocations
18:     ac ← availableContainers(c_size)
        // Get a random container of ac
19:     c_rand ← RAND(ac)
        //Switch a random container of cp with c_rand
20:     cp_[rand] ← c_rand
        return cp
21: function COSTS(cp)
22:     weight_d ← 0.5
23:     weight_c ← 0.5
24:     C_Closeness ← COSTCONTAINERCLOSENESS(cp) * weight_c
25:     C_DataLoc ← COSTDATALOCALITY(cp) * weight_d
26:     C ← C_Closeness * C_DataLoc
        return C
```

a normalized number between 0 and 1 that is compared with a random number between 0 and 1. This allows in some cases to choose a worse solution and get out of a local maxima. However, when ap gets smaller, which is the case when T becomes low or C_{new} is lower compared to C_{old}, the chance that a worse solution is selected decreases. Therefore, the chosen placement is more likely to stay in the last iterations and the chance of accepting placements with higher costs decreases.

The cost function consists of two parts. $C_{Closeness}$ describes the container closeness costs and $C_{DataLoc}$ the data locality costs. The total costs are calculated by multiplying both costs. Additionally, both costs are weighed with $weight_c$ and $weight_d$. Per default we weighted both with 0.5, however, it is important to emphasize that these weights can be configured individually for each job. The cost function is a major component of the algorithm, and thus is described in the next sections in detail.

6.3.2 Placing Containers Close Together

This section describes the container closeness term of our network-aware container placement model. Its goal is to place containers of a job close together with a low number of network hops in between them to reduce network demand. Especially on the core network layer, in which traffic aggregates more as multiple nodes share these links. However, it is important to emphasize that in some cases it is not beneficial to place all containers of a job dense on a small group of nodes. This is because containers in resource management systems such as YARN are instantiated without strict resource isolation. Hence, it is possible to oversubscribe and take advantage of temporarily unused resources of other colocated containers. However, dataflow task often stress the same resources at the same time, mainly due to their data parallelism execution concept. For instance, map tasks are often CPU-bound. When a map phase of a distributed dataflow job starts and all containers are placed on a few nodes, all containers compete for these few nodes CPUs at the same time. In this case, the advantage of oversubscription is invalid, as there is one single dominant resource that can slow down a job's execution time significantly. Therefore, instead of placing all containers of a job densely on a small group of nodes, it is more beneficial to balance them on a group of nodes that are close to each other in terms of network hops and leave some container slots free for job colocation and interference. The container closeness term $C_{Closeness}$ of our cost function covers both, network hop costs C_{Hops} and balance costs $C_{Balance}$.

$$C_{Closeness}(cp) = C_{Hops}(cp) * C_{Balance}(cp) \in [0,1] \tag{6.1}$$

In hierarchical multi-path networks, different container placement results in different involved communication paths. In addition, reducing communication over network

links on the core layer is beneficial in a oversubscribed and blocked network, because the available bandwidth between nodes within a rack is higher than the bandwidth between nodes across racks [32]. For instance, placing a jobs' containers on a single rack involves only the single top-of-rack switch, thus all traffic is kept on rack level and network congestions on the core network can be avoided. Therefore, it is preferable to place a job and its containers close to each other with a small number of links on the core network involved.

In order to determine the container closeness hop costs C_{Hops} of a container placement cp, we calculate and rate all network hops between the involved containers C. In particular, we determine and sum up each shortest path sp between all container pairs $\sum_{i=0}^{C}\sum_{j=i+1}^{C} sp(c_i \rightarrow c_j)$. For instance, when a container pair is host on the same node the hop count between both is 0. When a container pair is host on two different racks in a hierarchical fat-tree topology the hop count between both is 3, as three switches need to be passed. As a consequence, the cost function weights involved links on the core layer higher. In order to normalize the costs between 0 and 1, we take the current network topology $diameter$ as the max value, which is the maximum hop count between two nodes. Formally, the container closeness hop costs C_{Hops} are determined as:

$$C_{Hop}(cp) = \frac{\sum_{i=0}^{C}\sum_{j=i+1}^{C} sp(c_i \rightarrow c_j)}{\sum_{i=1}^{C}(C-i) * diameter} \in [0,1] \tag{6.2}$$

Containers are executed with no strict resource isolation, and thus, they compete for shared node resources such as CPU, disk and network Input/Output (I/O). The advantage of this method is based on the fluctuating resource demands of long-running analytics jobs. By running multiple jobs colocated in this way, it is possible to increase resource utilization and overall throughput due to statistical. At the same time, different tasks utilize different resources to different amounts. Sorting operations, for example, often require lots of memory whereas map tasks can, for instance, be mainly CPU-bound. Since dataflow systems are based on data parallelism, its tasks and containers execute the same program logic in parallel and thus tend to stress the same resources at the same time. For this reason, resources of a node are ideally shared by containers of different jobs. In other words, containers of a job should not be placed on just a few nodes. As a consequence, we define a container balance term that increases the costs when too many containers of a job

are placed on the same node. Combining this with the previously described hop costs, a good placement with low costs occurs, when a job's containers are placed in one rack and balanced across the available nodes in this rack.

In order to determine the container closeness balance costs $C_{Balance}$, we first determine all nodes N that cover at least one container of the placement cp. The number of containers that a single node covers is specified as n_{c_n}. The maximum number of containers a node can host is specified as n_{max}. When an jobs' container makes up to more than the balance threshold $bt = 0.75$ of the containers of a node, the cost increases. In order to normalize the costs between 0 and 1, we take the number of nodes hosting at least one container N as maximum value. Formally, the container closeness balance costs $C_{Balance}$ is determined as:

$$C_{Balance}(cp) = \frac{\sum_{i=0}^{N} 1, \text{if } \frac{n_{c_i}}{n_{max}} > bt}{N} \in [0,1] \tag{6.3}$$

6.3.3 Placing Containers Close to Input Datablocks

This section describes the data locality term of our container placement model. The goal is to place containers close to their job's input data to allow more local reads from the distributed file system. Most frameworks try to achieve data locality by placing source tasks on top of input data when possible. This can reduce network traffic and improve job execution time. Reading blocks locally is only constrained by the disk read speed, not additionally by the network throughput. Therefore, our goal is to place containers so that the jobs can explore data locality.

First, we determine the datablock cover ratio $C_{dbCover}$ of a container placement cp. In particular, we determine how many datablocks db_i of a job are covered by the container placement. Therefore, we loop over every datablock db_i, and check if there exists at least one replica of the datablock $db_{i,rep}$ that is stored on the same node as a container of the container placement cp. If this is not the case, the costs are incremented. In order to normalize the result between 0 and 1, we determine the ratio between covered blocks and total datablocks. Formally, the datablock covered ratio $C_{dbCover}$ of a container placement cp is defined as follows:

$$C_{dbCover}(cp) = \frac{\sum_{i=0}^{db} 1, \text{if } \nexists\, node(db_{i,rep}) \in cp}{db} \in [0,1] \tag{6.4}$$

Second, we determine the datablock replica cover ratio $C_{dbRepCov}$ of a container placement cp. In particular, we determine how many datablock replicas the placement cover in total. As blocks are replicated to provide fault-tolerance, placements can cover multiple replicas per block. Covering more than one replica introduces degrees-of-freedom for the framework's scheduling that often has to satisfy other constraints. When a node that stores a datablock replica $db_{i,rep}$ is not hosting any container of the placement, the costs increase. In order to normalize the result between 0 and 1, we determine the ratio between covered datablock replicas and total datablock replicas. Formally, the datablock replica cover ratio $C_{dbRepCov}$ of a container placement cp is defined as follows:

$$C_{dbRepCov}(cp) = \frac{\sum_{i=0}^{db} \sum_{j=0}^{rep} 1, \text{if } node(db_{i,j}) \notin CP}{db * rep} \in [0,1] \tag{6.5}$$

The data locality cost $C_{DataLoc}$ covers both previously described terms. However, covering all blocks is more important than covering many replicas. Therefore, the ratio of blocks covered C_{dbCov} is given more weight than the amount of replicas covered $C_{dbRepCov}$. Formally, the costs are defined as:

$$C_{DataLoc}(cp) = \frac{2\, C_{dbCov}(cp)}{3} * \frac{C_{dbRepCov}(cp)}{3} \in [0,1] \tag{6.6}$$

6.4 Evaluation

This section describes the evaluation of our container placement approach on a 8x8 core cluster organized in four racks using a fat tree topology. First, we provide details about our experimental setup and testbed. Afterwards, we describe our workload that consists of two iterative algorithms, K-Means Clustering and Connected Components, as well as two different TPC-H benchmark queries. Furthermore, we submit a number of concurrent Flink jobs, reflecting a high and mid cluster utilization. Afterwards, we present results of our experiments.

6.4.1 Cluster Setup

The evaluation took place on an eight node cluster, in which each node is equipped with eight cores, 32 GB of RAM, a single SATA disk and a 1 Gbps Ethernet network interface. The eight worker nodes are organized in four racks, so each rack consists of two nodes. The nodes are connected through a fat tree topology with two switching elements on the core and four on the edge layer. All switches (HP ProCurve Switch 1800-24G) are SDN capable and support OpenFlow 1.1.

Figure 6.4 gives an overview of our testbed. It is important to emphasize that we configured the switches into six different VLANs through port isolations. Furthermore, we connected the switches so that the VLANs form a hierarchical fat tree topology. VLAN 1.1 and VLAN 1.2 represent the two core switches and VLAN 2.1, VLAN 2.2, VLAN 3.1 and VLAN 3.2 represent the edge switches. So each host pair in different VLANs needs to communicate over one of the core switches VLAN 1.1 or VLAN 1.2.

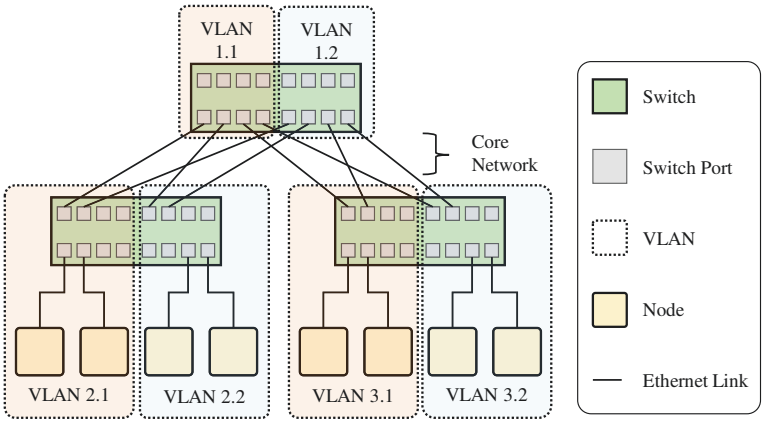


Figure 6.4: Overview of the cluster testbed.

We used an additional node as the master node for the cluster. This node manages the data storage and computing functionalities. Therefore, it runs YARN's ResourceManager, HDFS's Name Node and OpenDaylight as an SDN network controller. In addition, NeAwa's placement core-component to calculate a good container placement is hosted here. The remaining eight slave nodes are responsible for storing the data and running the workloads within containers. Therefore, each node runs YARN's Node Manager and HDFS Data Node. In our experiments all workloads run within containers with 1 vcore and 3 GB memory. Thus, we were

able to run 64 containers at the same time. In terms of software all nodes run Ubuntu 14, Apache Hadoop 2.7 and Apache Flink 0.9.

Since our testbed has only a few nodes, yet our approach targets hierarchical networks with hundreds or thousands of nodes of which network traffic aggregates on the core network, we simulated the targeting environment by shaping the network interfaces of the core network. Assuming $p_{total} = 48$ port switches available on the rack level with $p_{bw} = 1$ Gbps available per port and a blocking factor of 1:5 [40] ($bf = 0.16$), we have $p_{up} = 8$ ports with collectively 8 Gbps up-link bandwidth available between the core switches. The remaining $p_{down} = 40$ ports of the top-of-rack switches are available for hosts. In our testbed, we have $n = 2$ nodes per rack instead of 40, therefore, we have only a twentieth of possible nodes per rack. We define this as shape ratio $sr = \frac{nodes}{p_{down}} = 0.05$. Based on this, we shape the total available core bandwidth c_{bw} between the top-of-rack switches and core switches by $c_{bw} = P_{total} * P_{bw} * bf * sr = 400Mbps$. Therefore, each up-link port p_{up} has a max bandwidth of $\frac{c_{bw}}{p_{up}} = 50$ Mbps, and each top-of-rack switch has an up-link with 200 Mbps to each of the two core switches in our fat tree topology.

6.4.2 Jobs and Workload Description

This section introduces the workload that was used for evaluation. It consists of standardized benchmarks and productively used data analytics algorithms. The dataset size ranges from 8 GB to 250 GB per job. Furthermore, the jobs are diverse including relational database queries, machine-learning, and graph processing jobs. All jobs are implemented and executed in Flink.

Table 6.1 gives an overview of all four jobs that were used showing the algorithms, datasets, parameters and cost function weights. The cost function weights are calculated for each jobs individually by a certain calibration experiments, which is described in detail after the benchmark job description.

TPC-H [131] defines standardized benchmark queries for databases and transaction processing systems. We chose the TPC Benchmark suite H (TPC-H) Query 3 and 10. Both are business oriented ad-hoc analytical queries that examine large volumes of data. For each job, we generated a unique input dataset using the official TPC-H data generator.

Algorithm Name	Dataset Size	Algorithm Parameters	Closeness Weight (w_c)	Data Loc. Weight (w_d)
K-Means Clustering	8 GB	8 clusters, 10 iterations	0.7	0.3
Connected Components	25 GB	5 connectivity	0.7	0.3
TPC-H Query 3	250 GB	-	0.2	0.8
TPC-H Query 10	250 GB	-	0.3	0.7

Table 6.1: Overview of NeAwa's Benchmark Jobs.

K-Means Clustering [132] is a compute intensive data processing algorithm, which is used in the area of Machine Learning. It is an iterative algorithm that groups a large set of multi-dimensional data points into k distinct clusters without supervision. For our evaluation, we generated eight random fixed centers and 600 million points, resulting in approximately 8 GB input data.

Connected Componenents [133] is an iterative graph algorithm that identifies the maximum cardinality sets of vertices that can reach each other in an undirected graph. We used a label propagation-based implementation due to its better scalability and parallelization capability [134]. For our evaluation, we used a Twitter dataset with around 25 GB input data [135].

Cost Function Calibration. NeAwa is based on a weighed cost function with the two weights for container closeness (w_c) and data locality (w_d). In order to derive good initial weights for each job, we execute a calibration experiment with varying weights. Starting with $w_c = 0$ and $w_d = 1$, in each run r, we increase $w_{c_r} = w_{c_{(r-1)}} + 0.1$ and decrease $w_{d_r} = w_{d_{(r+1)}} - 0.1$, until both weights are 1 and 0. As the final initial weights for a job, we chose the two pairs of a run (w_{c_r}, w_{d_r}) with the lowest execution time. The results for each job are shown in Table 6.1. It is important to emphasize that we run all experiments in standalone without interference to other jobs. Besides that, the weights depend on the cluster setup as well as job specific characteristics such as the amount of allocated resources, input dataset size and its block partitioning.

Workload Description We defined two different workloads of terms of cluster utilization. A high cluster utilization using nearly all available resources and a mid utilization using the half. The reasons for that is twofold. First, the network is no bottleneck when it is not shared by multiple jobs, as all bandwidth is assigned to this job. Second, interference between different jobs is important in data analytics platforms, as they are executed with weak isolation to allocate or deallocate more local resources on demand.

High cluster utilization. For the first experiments, we used 60 of 64 available cores. We submitted five applications, each using 12 containers, in which one was used for the application master and eleven for Flink tasks. Each container allocated 3 GB of RAM and 1 vcore. We run five different experiments under this utilization. Four experiments containing only one specific type of job. Another mixed experiment with five job types of algorithms. For instance, in one experiment we run five K-Means jobs at the same time. In the mixed one, we execute a K-Means, Connected Components, TPC-H Query 3 and two TPC-H Query 10 at the same time. In all experiments, we submitted the five applications with a delay of 10 seconds in-between. For each job we generated a separate dataset to have a different location for the input datablocks of different applications. For instance we have a total of 40 GB input data for the K-Means clustering workload and 125 GB for the Connected Components workload.

Mid cluster utilization. For the second experiments, we used 36 of 64 available cores. In this experiment we submitted three applications, each using 12 containers. Each container allocated 3 GB of RAM and 1 vcore. Similar to the high cluster utilization scenario, we execute five different workloads under this utilization. Four workloads containing only one specific type of algorithm. Another mixed workload with three different types of algorithms, in which we chose K-Means, Connected Components and TPC-H Q3. For each job we generated a separate dataset to have different locations for the input datablocks.

6.4.3 Results of Different Workload Scenarios

This section presents the results of the previously described workload scenarios. We compare execution times with NeAwa's placement with Hadoop YARN's default container placement. Table 6.2 gives an overview of the results.

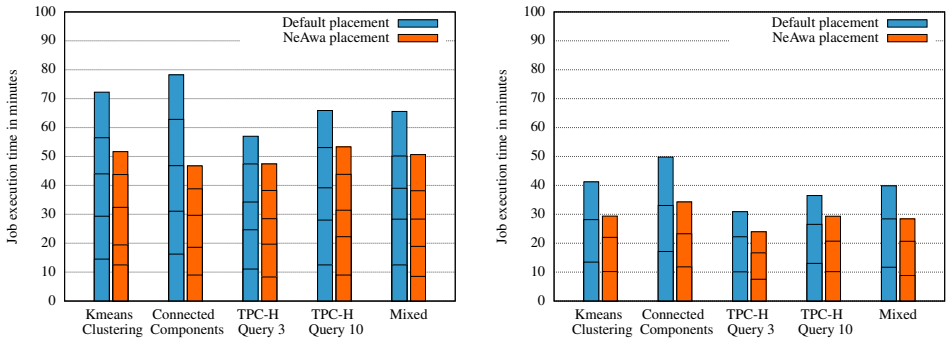
Cluster Utilization	Workload	Speed-Up
High (60 of 64 cores)	K-Means	39.6%
	Connected Components	67.9%
	TPC-H Q 3	21.6%
	TPC-H Q 10	21.7%
	Mixed	26.8%
Mid (36 of 64 cores)	K-Means	41.2%
	Connected Components	45.1%
	TPC-H Q 3	16.0%
	TPC-H Q 10	17.3%
	Mixed	25.0%

Table 6.2: Overview of the results comparing NeAwa and Hadoop's default container placement.

High cluster utilization results. Figure 6.5a shows the results of the high cluster utilization workload, in which we execute five jobs at the same time that uses nearly all available cluster resources. In the bar chart, each block represents the execution time of a job, stacked to sum the execution time of all five jobs. The speed-ups are ranging between 16.0% and 45.1%. For the K-Means workload, in which we executed five different K-Means jobs at the same time, the runtime decreases by 39.6%. For Connected Components 67.9%, for TPC-H Query 3 22% and for TPC-H Query 10 17.3%. For the mixed workload the speep-up was 26.8%, in which we executed one K-Means, one Connected Components, one TPC-H Query 3 and two TPC-H Query 10 at the same time. Figure 6.6 shows the execution time of the mixed workload in a time-sequenced view.

Mid cluster utilization results. Figure 6.5b shows the results where each block represents the execution time of an application, stacked to sum the execution time of all three applications. The speed-up for the K-means clustering workload was 41.2%, for the Connected Components workload 45.1%, for the TPC-H Query 3 16.0%, for the TPC-H Q 10 21.7%, and for the mixed workload 25.0%. The mixed workload consisted of one K-Means, one Connected Components and one TPC-H Query 3 job.

Discussion. Comparing workloads with different utilizations, the speed-up is higher when the cluster is more utilized. This is because, the network becomes shared by more jobs and thus, it becomes a sacred resource. It is important to emphasize that we simulated a blocking factor with traffic shaping on the core



(a) Workloads consisting of five jobs.

(b) Workloads consisting of three jobs.

Figure 6.5: Comparing NeAwa and the default container placement implementation of Hadoop and Flink with different workloads.

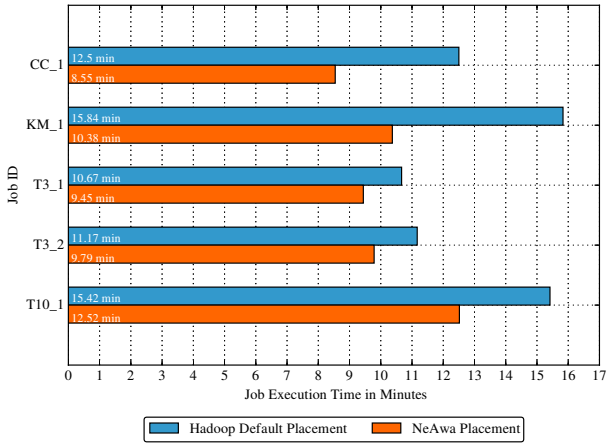


Figure 6.6: Runtimes of a mixed workload that consists of five different jobs scheduled simultaneously and utilizing all compute resources.

network. Even having considered this, NeAwa should increase the performance in productive clusters, as these are often deployed with a blocking factor, too. Another finding is that iterative machine learning and graph processing jobs, which are in our experiments K-Means and Connected Components, gain more performance optimization, compared to relational database queries such as TPC-H. This is because containers of this class of jobs are scheduled closer to each other, compared to the default placement scheduling, and thus, can exchange intermediate results more locally. Furthermore, the testbed consists of only eight nodes. Hence, the input datablocks are locally available in almost every placement. However, when scaling up to more nodes than that, the performance may increases even more significantly.

Chapter 7: Data Retention Placement

Contents

Endolith [142] is a data placement framework for improving data retention in data analytics platforms. In general, it provides a tamper-proof tracking and management of shared datasets and analytic results. Endolith does this by automatically generating metadata describing changes of these shared data and storing it immutably on a blockchain network. These sensitive metadata are only accessible for chosen users and through unchangeable program code and functions, as they are deployed and executed as smart contracts on a blockchain network. Therefore, Endolith supports storing datasets as well as data analytics results long-term reliably, because a tamper-proof status of these data is stored on the blockchain that can be used for validation, auditing and change tracking. As Endolith is based on a smart contract-based blockchain, it does not rely on a central trust authority. Furthermore, the data analytics platform provider that integrates Endolith and its users interact with each other in a fully decentralized and automated fashion.

This chapter is structured as follows. First, it describes the concept of Endolith long-term data retention approach. The chapter then presents the system overview. Afterwards, the smart contract, its deployment and execution mechanism are discussed. This is followed by a description of Endolith's workflows. Finally, a performance evaluation of Endolith at scale showing its feasibility is reported.

7.1 Improving Long-term Data Retention

Data Retention is the process of storing data securely with the purpose of using it later again in the exact same state as it was stored. However, this is difficult to achieve in a shared data analytics platform with many users sharing one storage system. The problem with data retention is not necessarily the data storage itself. Data backups procedures, disk and data stripping are widely used techniques to deal with it. The problem comes in ensuring the data's integrity. For instance, proving that data stored a long time ago has not been changed or become corrupt, and if it happened, tracking what has changed, by whom and when.

Most storage systems in data-analytic platforms provide a log file that contains all operations happening on files. This log file is often used for tracking and audit purposes. However, this information is for due to privacy reasons only accessible for a few people like system administrators and applications that focus on system security features like intrusion detection. Also, it is difficult to prove that this log file was not changed by an unauthorized user, too. Therefore, from a user's perspective, it is challenging to validate if a file's content in a shared remote storage system is changed or got corrupted.

This motivated us to design a novel long-term data retention method by supporting tamper-proof data tracking and verification using a smart contract-based blockchain. In particular, the method follows the DApp [143] principle, where the logic of the method is executed as smart contracts on a decentralized peer-to-peer blockchain network. Therefore, it operates autonomously without the need of a central controlling entity. The method's input data, which is metadata describing the current state of a file, is automatically generated and placed on the blockchain when a file operation occurs in the storage system. Users interact with well-defined smart contract functions to access these data.

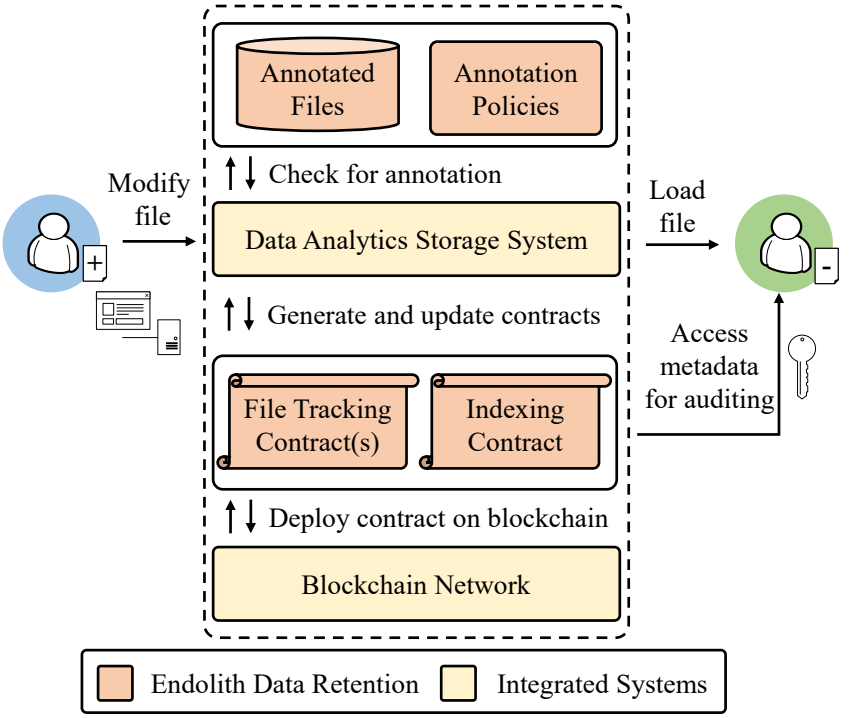


Figure 7.1: Improve data retention by storing selected file metadata on the blockchain and offering the user a functionality to evaluate files against it.

Figure 7.1 shows the method of our smart contract-based blockchain approach for improving data retention on data analytics platforms. When a file is modified on the data analytics storage system, it is validated if the file is annotated for tracking. If this is the case, then, Endolith long-term data retention takes place and unique metadata and attributes of the file's new state is automatically generated.

Afterwards, either a new dedicated smart contract for that file is automatically generated and deployed on the blockchain network, or an existing smart contract is updated with the file's metadata. The file tracking smart contract stores all necessary data for file validation and tracking. This includes the current and all past file hashes of the tracked file as well as its modification time and user. A file tracking contract belongs to exactly one file.

After a file tracking contract is successfully deployed, the unique smart contract address provided by the blockchain network is stored in a global index smart contract. This index smart contract maintains all file Uniform Resource Identifiers (URIs) and smart contract address relations.

In addition, a smart contract consists of an access control mechanism based on public key cryptography. Only selected people based on their key are allowed to execute functions, for instance, to add and access data of the smart contract.

Later, when the same or another user wants to validate their file, he or she can look up the metadata on the blockchain via the dedicated smart contract for auditing processes. Based on this, it is possible to ensure file integrity without the reliance on a central trust authority. In particular, we use the *MD5*[10] algorithm to produce a 128-bit hash value of a file. This checksum is used to verify data integrity by comparing the local and remote blockchain hash of a file.

It takes a certain time until a transaction or smart contract creation is mined and confirmed on the blockchain. Also, writing a transaction is expensive, due to the proof-of-work consensus algorithm that is used in most blockchain networks. Therefore, Endolith is best applied for files that are infrequently modified and where user response time for creating or modifying a file is not critical. An example for this class of files are archive files.

7.2 System Overview and Integration

This section provides the system overview of Endolith, as shown in Figure 7.2. It is integrated with a storage system and smart contract-based blockchain network, and acts as mediator between both. Endolith monitors annotated files and translates operations on it to blockchain transactions. In addition, it provides auditing functionalities to its users based on smart contracts that operate on these transactions. The remainder of this section describes Endolith's components in detail.

Storage User. The user stores its files or results of a data analytics job on a remote storage system. Files can be annotated to be tracked with Endolith by the user or the system itself based on an automated annotation policy. Similar to public-key encryption, only users whose public key is defined in the file tracking smart contract can execute smart contract functions with their private key. Thus, only these users can access the data stored on the blockchain network. Other users with no access to the smart contract can only see data stored in the smart contract and its code in form of raw binary data within transactions on the blockchain network.

[10] `https://tools.ietf.org/html/rfc1321`, , accessed 2018-08-06.

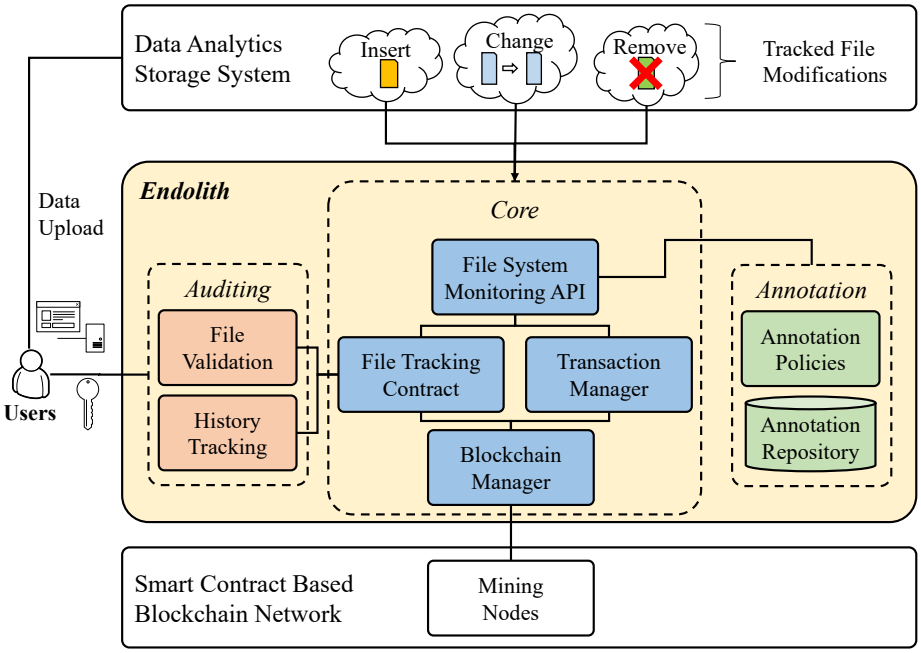


Figure 7.2: Overview of Endolith and its components.

File System Monitoring API. The Application Programming Interface (API) provides methods for adding information about file creation, changes, and deletion to Endolith. Additionally, it can be used to add information in the other direction from Endolith to files. For instance, to add the smart contract address as metadata to its corresponding file, which allows an easier association between both. It can also be used by a storage provider to automatically monitor their storage systems by using hooks that listen to file operations and then automatically use the API to forward the collected data to Endolith. In our prototype, we integrate Endolith with our monitoring system Freamon that monitors file operations of HDFS. In addition, Freamon collects necessarily metadata and attributes of the annotated files such as the file hash, user name, and modification time.

File Tracking Contract. We designed a File Tracking Contract template of which Endolith automatically deploys an instance of any annotated file. An instance is responsible for tracking all changes of exactly one file. Due to scalability requirements for handling a large volume of files and changes, it is not feasible to store all metadata in one global smart contract. This is because it would take a lot of time to index that contract and there is a limitation in terms of size for a smart contract. After a new file is stored and its related smart contract is successfully deployed, its

smart contract address is attached to the file and stored in a global index contract to find the corresponding file on the blockchain network for later changes on that file.

Transaction Manager. The transaction manager deploys the previously described file tracking contracts on the blockchain network. Additionally, it converts data provided by the file system API into a smart contract executable format. In the other direction, it translates transaction receipts to a file system API capable file format. Additionally, it buffers transaction if needed. This is because it takes time until a transaction becomes available on the blockchain. Instead of waiting until the transaction is available and verified, we collect the status and periodically check if the transaction is successfully written to the blockchain. If a transaction is rejected by the blockchain, the transaction is submitted again. Following transactions that effect the same file are queued in a first in, first out queue and submitted after the ancestor was successfully written to the blockchain.

Blockchain Manager. The manager acts as gateway to the smart-contract-based blockchain network and provides access to it. This component participates in the blockchain network and thus, has a local copy of the blockchain. However it is decoupled from the mining network, thus not acting as a miner to perform the proof-of-work algorithm. The reason for that is to save compute resources for Endolith's main tasks. In particular, the blockchain manager has two tasks. First, sending contract creation and execution transactions to the blockchain network, where they are mined and executed. Second, some smart contract functions are directly executed on the manager, as they do not modify the global state of the blockchain. The latter functions operate on the local copy of the blockchain, thus returning fast results.

Annotation Repository and Policies. The repository consists of a collection of all known annotated files. An annotation can either be done directly by a user or an automated policy. The latter can be based on its age and usage frequency, similar to policies used in automated tiered storages that assign data to a specific archive data tier. In addition, the unique smart contract address per file is stored to a global index smart contract. This index smart contract maintains all annotated file URI and smart contract address relations. It serves as fall back mechanism, as there is a possibility that the contract ID directly attached as metadata to a file may get accidentally or maliciously deleted or changed.

File Validation and File History Tracking. Both auditing functions are embedded in smart contracts and are executed locally on the node running Endolith. Also, they are executed on an up-to-date copy of the blockchain and do not change the state of the contract, and thus can be executed without the need of writing a transaction on the blockchain. File Validation ensures that the file is correct. The user can validate if the local and remote files are the same, based on the local and remote baseline hash stored on the blockchain. This can be useful after uploading or downloading a file from the storage system. File History Tracking refers to the process of tracing and recording files with explanation of how it got to the present state. Therefore, every modifications on selected files is immutably recorded on the blockchain network. Endolith uses the information to provide track change history of a file record.

7.3 Smart Contract Blockchain-based File Tracking

This section describes the Endolith smart contract blockchain-based file tracking mechanism. First, we present Endolith file tracking smart contract template. Afterwards, we discuss the transaction manager in more detail, which deploys and executes the smart contracts to the blockchain network.

7.3.1 File Tracking Contract Template

For any file that is annotated for data retention, a corresponding File Tracking Contract is generated and deployed on the blockchain network through Endolith's Blockchain Manager. Each contract is derived from a generic template. Table 7.1 gives an overview of this template and describes its parameters and functions in detail. A File Tracking Contract belongs to exactly one file, and thus, for every annotated file exactly one contract exists on the blockchain network. It is important to emphasize that a file URI of a File Tracking Contract can change due to file renaming or moving. Therefore, each unique smart contract address is attached to the corresponding file as a metadata attribute and to a global index smart contract that maintains all file URIs and smart contract address relations. Afterwards, every file modification is stored on the blockchain using its corresponding smart contract, which is identifiable by the smart contract address.

Variable	Description
contractAddr	A blockchain-wide unique address to load and execute a smart contract.
owners	A list that contains all public keys of users that can execute the smart contract functions and constructor.
contractCreation	Timestamp of the contract creation.
lastOperation	Timestamp of the last file operation.
currentHash	A cache variable that stores the last known hash to speed-up requests for validation.
currentFileURI	The current URI, which can be changed when a file name, path, or the storage system changes.
operations	A list of all tracked file operations and its metadata and attributes. This includes all file hashes of the tracked file, its timestamp, operation type, user, fileURI and block locations.
Functions	**Description**
FileTracking	When constructing an file contract, the corresponding file URI and hash is required.
fileOperation	This function is executed when an operation on the corresponding file occurs. Thus, it requires operation type, files path and the new file hash as input. All new values are pushed onto the list of operations. This function invocation changes the global state of the contract and results in a transaction.
validate	The validate function verifies that any given hash equals the current value stored on the blockchain and returns a boolean value accordingly. The validate call does not alter the state of the EVM and therefore does not require a transaction to be sent and can be performed locally.
getEntryAtTime	Any hash for any point in time since contract creation can be retrieved and used for history tracking and file validation.
modifier onlyBy	This modifier is used to ensure that an annotated function can only be executed from an address stored in the owners list.

Table 7.1: Overview and explanation of Endolith's file tracking smart contracts variables and functions.

7.3.2 File Tracking Transaction Management

The Transaction Manager is responsible for deploying and executing Endolith's smart contracts to the blockchain network. An important characteristic here is that it takes up to several seconds until a submitted transaction for contract creation or execution is mined and confirmed by the blockchain network. We estimate the confirmation time ct of a transaction tx by:

$$\Delta ct_{(tx)} = (\Delta bt * c) + \Delta \frac{bt}{2} \tag{7.1}$$

Let Δbt be the average block time that the blockchain network takes to generate a new block. c is the number of subsequent blocks that must confirm that the transaction is valid and is part of the blockchain. $\Delta \frac{bt}{2}$ is the estimated pickup time between transaction submission and transaction being mined in a block, assuming the transaction gets mined in the next mined block. On the Ethereum main net the average block time $\Delta bt = 15$ seconds[11]. The Ethereum whitepaper suggests to wait $c = 7$ block confirmations [143]. As a result, we estimate the conformation time of a transaction $ct_{(tx)} = 112.5$ seconds.

Taking this high conformation time into account, Endolith fits only for files that are less frequently modified such as archive datasets. However, to overcome this limitation that results in a high overhead response time when writing a file, Endolith caches transactions until they are included into the blockchain and enables to work on them during this time. In particular, Endolith's Transaction Manager holds a list of all pending contract transactions and periodically checks if the creation is successfully deployed. In order to allow modification on that file during its waiting time, all subsequent changes are stored in a dedicated waiting queue for that file. Once the contract creation transaction is confirmed, the contract address is received and used to write all updates to the corresponding tracking contract. We implemented this as a first in, first out queue, when a transaction is successfully written and confirmed, the next transactions are submitted. In blockchain networks the order of transactions is important. For instance, to avoid replay attacks, in which a valid transaction is maliciously over and over to the network. Therefore, we buffer and count transactions in memory and wait until the predecessor was successfully confirmed. It is important to emphasize that Endolith uses one central wallet for Endolith that allows to submit a pool of transactions. To ensure transactions are processed only once, the wallet has a transaction counter called 'account nonce'. It describes the number of transactions sent from an account. If a transaction is submitted with an account nonce that has already been transacted, the transaction will be rejected by the network. In such a situation, or if a transaction fails due to other reasons, it is possible to reconstruct the nonce based on buffer and last known transaction receipts.

[11] `https://etherscan.io/chart/blocktime`, , accessed 2018-08-06.

7.4 Placement and Validation Workflow

This section describes the placement and validation workflows of Endolith for file tracking, validation, and history tracking in more detail.

File Tracking. The workflow is shown in Figure 7.3 and consists of the following steps. *(1)* The user or annotation policy annotates a file for long-term file tracking. *(2)* A dedicated smart contract for that file is automatically generated based on Endolith's file tracking template. *(3)* The generated file tracking contract is submitted to the blockchain network. *(4)* A unique smart contract address is returned after it is successfully deployed on the blockchain. This contract address is attached as meta data to the file residing in the storage system to make a connection between the smart contract and the file. In addition, it is stored in a global indexing smart contract for long-term. *(5)* The file hash is calculated using a cryptographic one-way hash function. Additionally, unique meta data and attributes like user name, modification time, and block location is collected from the storage system. It is important to emphasize that hashing is not done within the smart contract on the blockchain. This would be expensive and even impossible for large files, due to the file data that needs to be sent to and processed on the blockchain network. *(6)* All data is parsed to the corresponding file contract, which is identifiable by the contract address attached as metadata to the file. *(7)* Endolith receives a receipt confirming that the transaction was successfully written to the blockchain. *(8)* Afterwards, once the transaction is successfully written to the blockchain, the user is notified and, if additional queued transactions for that annotated file exist, they are submitted to the blockchain network. Step *(1 - 4)* are only done once, when the file is annotated. Steps *(5 - 8)* are repeated every time a file changes. In order to reduce the waiting time until a transaction is available on the blockchain, subsequent transactions are queued in the transaction manager.

File Validation. The workflow of validating a tracked file with Endolith is shown in Figure 7.4 and consists of the following steps. *(1)* The user requests a file from the file system. *(2)* The corresponding file tracking contract is loaded based on the smart contract address attached as metadata to the file. *(3)*Endolith requests the last known hash from the loaded file tracking contract. *(4)* The smart contract validate function is executed locally by Endolith on the up-to-date copy of the blockchain, to get the last known hash stored on the blockchain for that requested file. *(5)* The

hash of the file version stored in the storage system is calculated. *(6)* The current resulting hash value and the last value stored in the smart contract are compared to each other. If both values match, the file is presumed to be the same, because the probability of an accidental hash collision, which means that a file will hash to the exact value, is very small. *(7)* The user loads the validated file.

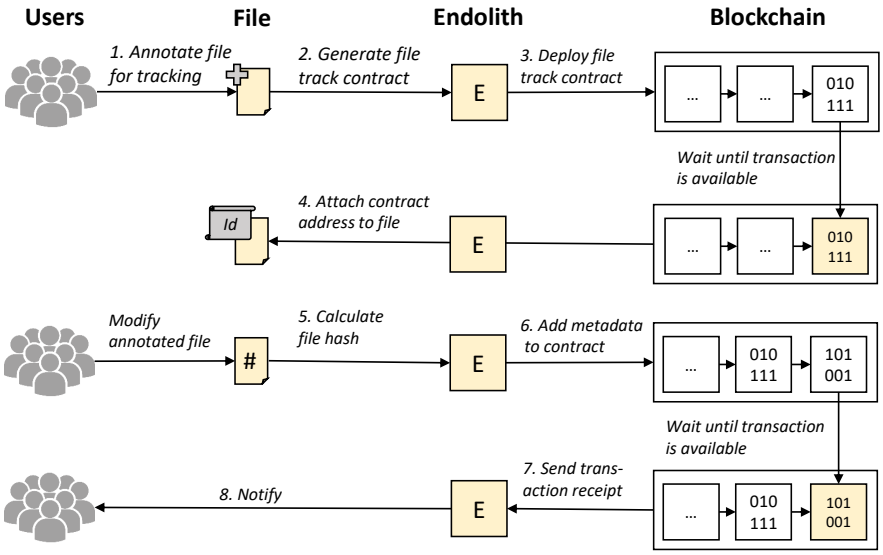


Figure 7.3: Workflow of annotating and monitoring with Endolith.

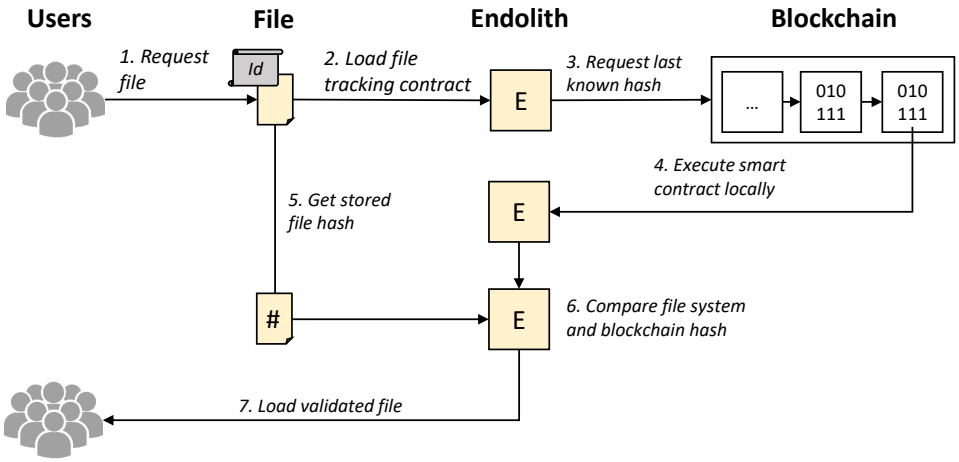


Figure 7.4: Workflow of validating a file with Endolith.

In addition, Endolith allows its users to validate their local copies of a file against the ground truth stored on the blockchain network without touching the remote file. Also, he or she can validate, if the local version existed at any time. Another

advantage of this hash-based file verification is its good performance without the need of comparing bit per bit and it is possible to compare files without making its content visible to anyone. Also the hash size is relatively small in size compared to the file size, and thus is suitable to be stored on the blockchain.

History Tracking. By storing the file hashes including additional file attributes maintained by the storage system, such as owner, time stamp, and URI, on a blockchain using dedicated smart contract, it is possible to guarantee and exactly determine when and how a file has being changed at any moment in time. If an already stored file is modified, the updated file attributes including the new file hash is appended to the smart contract and is stored in the blockchain network. As soon as the transaction is mined and confirmed, we can indisputably prove that the file record in the storage exists and securely track who changed a file when, because transactions on the blockchain cannot be modified or deleted. Additionally, all that functionality can be done without any third party interference.

7.5 Evaluation

This section presents the evaluation of Endolith. First, the benchmark setup and testbed is described. Afterwards, the overhead and costs of writing and reading a file with Endolith is presented.

7.5.1 Cluster Setup

All experiments were done using a 11 node cluster. Each node is equipped with a quad-core Intel Xeon CPU E3-1230 V2 3.30GHz, 16 GB RAM, and three 1 TB disks with 7200RPM organized in a RAID-0. All nodes are connected through a single switch with a one Gigabit Ethernet connection. Each node runs Linux, kernel version 3.10.0, and Java 1.8.0. 10 Nodes run Hadoop HDFS 2.7.1 with default configuration. One Node runs Endolith and Geth[12] version 1.7.1 as gateway to Ethereum's public testnet Ropsten[13]. In contrast to the ethereum main net, writing transactions is free on the testnet. Additionally, new blocks on the testnet are mined

[12] `https://github.com/ethereum/go-ethereum`, , accessed 2018-08-06.
[13] `https://github.com/ethereum/ropsten`, , accessed 2018-08-06.

with a less difficulty parameter, and thus are generated and validated faster. At the time of writing this paper, the block time on Ethereum main net is average 15 seconds. The testnet does not provide detailed historical data about the block time, our observations show a varying block time between 10 and 15 seconds. Furthermore, the geth client waits 5 blocks for confirmation, instead of 7 blocks, as suggested in the Ethereum whitepaper [143].

7.5.2 Benchmark Description and Results

The evaluation consists of three benchmarks. First, the response times of writing a single file and multiple files to test Endolith at scale are measured. Second, the response time or reading and evaluating are reported. We repeat all experiments seven times and report the median. Third, the additional costs of using Endolith are reported.

Writing Response Time. This benchmark measures the overhead in terms of response time when writing a file caused by Endolith. Therefore, we generate multiple random files with fixed file sizes ranging from 64 MB to 8192 MB by using dd, a command-line utility for Unix and Unix-like operating systems. Figure 7.5 reports the results of writing files with varying sizes. The blue bar represents the response time until the file is available in HDFS. The orange bar represents the additional response time until the metadata is collected, the contract is generated, deployed and confirmed by the Ethereum network.

The median response time for collecting the metadata as well as sending and confirming the contract transaction is 42.5 seconds. This response time is approximately the same, independent of the file size, because the opcode and input data for deploying a file tracking smart contract has approximately the same size independent of the file size. However, maximum and minimum values are far from each with 15.59 seconds and 105.8 seconds. This is because block time varies heavily on the testnet. Collecting the metadata and sending the contract creation took 0.7 seconds on the average. Transaction confirmation took most of the response time.

When the file size increases, the proportion of the overhead caused by Endolith decreases, because more time is spent on the file transfer. It is important to emphasize that Endolith allows the user to use HDFS during that time due to the Transaction

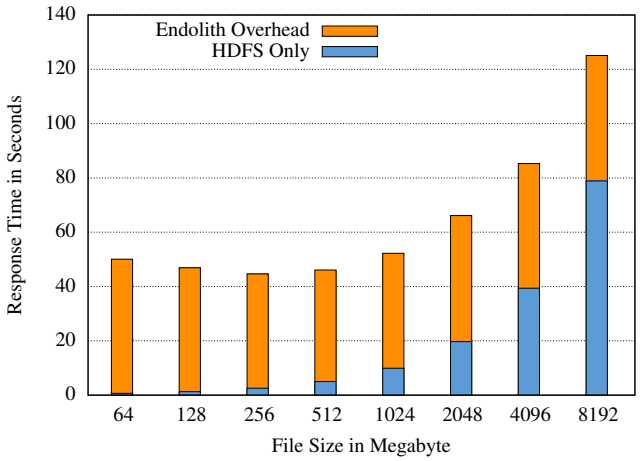


Figure 7.5: Response time of writing a file with varying size to HDFS with and without Endolith.

Manager that caches transactions until they are included into the blockchain and enables to work on them during this time. However, due to the long response time until a transaction is confirmed, Endolith is more suitable for archival data, which is modified rarely or not at all.

Figure 7.6 reports Endolith writing performance at scale. Therefore, we generate a new file every 15 seconds and every already stored file is changed after these 15 seconds too. As a result, the number of newly submitted transactions increases by 1 after every 15 seconds. For instance, after the first 15 seconds one new transaction was submitted, after 30 seconds two transactions (one contract creation for a new file and one for a file update), after 45 seconds three transactions (one contract creation and two file updates), and so on. By this, the load increases consequently while the test is running. In total, we ran the experiment for 3000 seconds. During that time we generated 20301 total transactions, as shown in the red graph. The blue graph shows the total number of confirmed transactions by the network. The gap between both lines indicates how long it takes until transactions are confirmed. It can be seen that it takes approximately the same time as in the response time benchmark, even when the system load and number of transaction increases.

Reading and Validating Response Time. Similar to the writing benchmark, we generate random files with varying sizes between 64 MB and 8192 MB. Figure 7.7 reports the results of reading and validating the files with varying sizes. Before

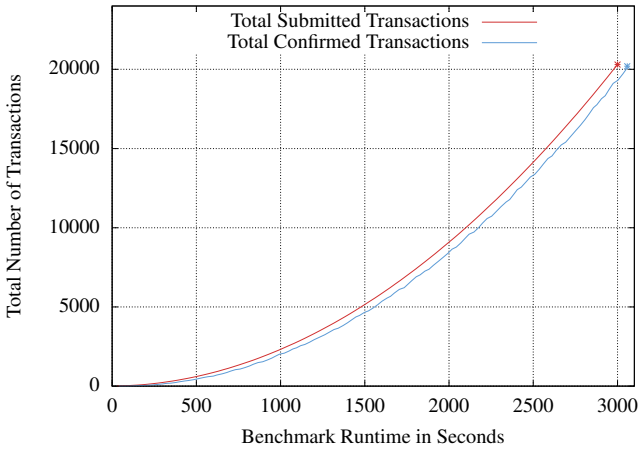


Figure 7.6: Evaluating Endolith write performance at scale by tracking and changing multiple files simultaneously.

reading the file with Endolith, we modify it 50 times to generate data within the corresponding file tracking contract. The blue bar represents the response time until the file is copied from HDFS to the local disk. The orange bar represents the response time until the local file is hashed and validated against the value stored on the ethereum blockchain. The overhead varies between 0.54 to 1.20 seconds. The red line represents the overhead in percent, which varies from 87.83% to 1.62%. When the file size increases, the overhead caused by Endolith decreases, because more time is spent on the file transfer. In comparison to the write response time, the reading and validating overhead is small. This is because the smart contract execution for retrieving the file hash can be done locally without writing any transaction on the blockchain network.

Costs of Creating And Modifying a File. This benchmark measures the cost of creating and modifying a single file in terms of gas [143]. Gas is the name for a special unit used in Ethereum. It measures how much work an action or set of actions takes to perform. Every operation that can be performed by a transaction or contract on the Ethereum platform costs a certain number of gas, with operations that require more computational resources costing more gas than operations that require few computational resources.

We generated a 1024 MB file, loaded the file to HDFS, and repeated both steps 10 times for overriding the file in HDFS. When a file is created, a corresponding

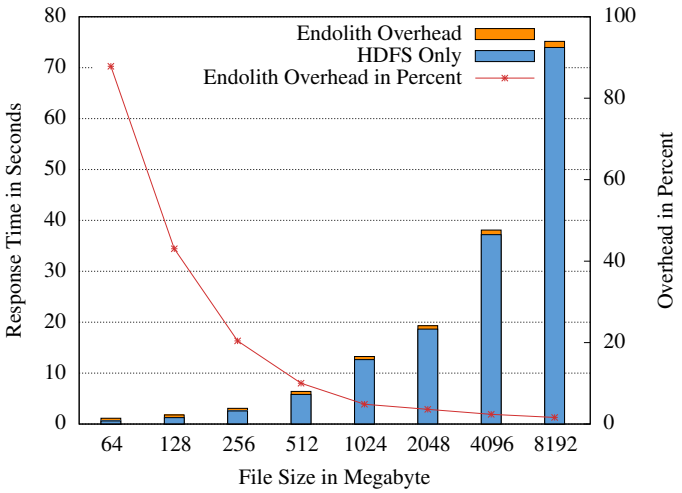


Figure 7.7: Response time of reading and validating a file with varying size from HDFS with and without Endolith.

smart contract is deployed that costs 18.901 gas. These are 0.062€ based on the gas price[14] and Ethereum-Euro exchange rate[15] on 2018-08-13. Every file modification costs constantly 2.632 gas, which are 0.009€. It costs the same amount, because independent of the file size, the opcode and input data deploying a file tracking smart contract or adding new data to an existing one has approximately the same size independent of the file size. However, users should be thoughtful which and how many files to track when using Endolith on a public network, because executing code on the blockchain generates costs. It is important to emphasize that reading and validating generates no costs in terms of gas usage, as the contract is executed locally based on a valid blockchain copy without the need for writing any transaction.

[14] `https://etherscan.io/chart/gasprice`, , accessed 2018-08-06.
[15] `https://trade.kraken.com/kraken/etheur`, , accessed 2018-08-06.

Chapter 8: Conclusion

This thesis presented methods that optimize the runtime performance of distributed dataflow applications running in shared data analytics platforms. In particular, two methods are presented that optimize the runtime performance of distributed dataflow applications through a better coordination between datablock and container placements. By this, runtime performance can be improved without allocating more containers and resources per job, increasing the amount of datablock replicas in the cluster, or changing the application itself. In addition, this thesis tackles the problem of long-term data retention by a method that allows to validate and track files based on metadata stored on a blockchain network. Besides the metadata, the validation and tracking logic is also immutable as it is implemented and executed as dedicated smart contracts on the blockchain network.

The thesis made contributions in three areas. The first contribution are two container and datablock placement methods to optimize the runtime performance of distributed dataflow batch applications. The first placement method, which is called CoLoc, places related datablocks and containers on a pre-selected set of nodes. By this, it improved the runtime performance of various dataflow applications in comparison to using Hadoop's default placement method by reading more input data from local disks and having more local inter-process communication between containers and connected dataflow tasks running inside it. The second placement method, which is called NeAwa, improved the runtime performance by approximating a good container placement using Simulated Annealing. Its cost function takes network distances between a job's containers and input datablocks as well as balancing and interference into account. The second contribution is a method called Endolith. It supports long-term storing of shared files by embedding metadata describing the lineage of data transformations on a blockchain through dedicated smart contracts. Users can use well-defined functionalities of these contracts to

validate data integrity. The third contribution is given by the specification of a dynamic data and resource management system to improve automation, runtime performance and data retention in data analytics platforms.

All methods of this thesis are implemented in a research prototype system and have been evaluated on a 64 node commodity cluster. In order to show runtime performance improvements, workloads consisting of different standardized benchmarks, datasets, and applications used in production are used. We use Flink as reference distributed dataflow system for our evaluation and different types of batch applications. The results are promising. CoLoc shows a reduction of execution times up to 31.19% depending on the job, workload, and number of colocated jobs. NeAwa shows a reduction of execution times between 25.04% and 26.81% for mixed workloads consisting of graph, machine learning and data-intensive relational queries. Endolith is based on Ethereum and is evaluated on its official testnet. Its performance evaluations demonstrate that the method can improve retention at scale and with low overhead in terms of additional response time.

Although this thesis already shows promising results for improving runtime performance and data retention in data analytics platforms, some interesting directions for further investigation exist. The placement strategies affect runtime performance differently depending on the type of application. Therefore, we would like to investigate more methods to automatically select the most appropriate strategy for upcoming applications. Furthermore, we used Flink as reference distributed dataflow system, allowing us to run a broad mix of applications. However, further evaluations with other non-distributed dataflow systems should be investigated to determine if the placement methods are applicable to them as well. Other directions can be derived from our method's limitations. One limitation of data and container colocation is that dataset and job dependencies need to be set manually. In order to decrease the required user intervention, we would like to combine the placement strategy with file access prediction techniques to identify datasets that are accessed and processed together automatically and re-balance the datablock distribution at runtime. Our blockchain-based data retention method is best applied for cold data, which is infrequently modified and whose user response time for creating or modifying a file is not critical. Therefore, we would like to find out how Endolith performs on a private blockchain with a lower block time. In addition, using a private blockchain allows being independent from price fluctuations and speculations.

Bibliography

[1] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google News Personalization: Scalable Online Collaborative Filtering," in *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pp. 271–280, ACM, 2007.

[2] S. Kedia, S. Wang, and A. Ching, "Apache Spark @Scale: A 60 TB+ Production Use Case." https://code.facebook.com/posts/1671373793181703/apache-spark-scale-a-60-tb-production-use-case/, 2016. Last Accessed: 2018-09-30.

[3] J. Wozniak, "The Architecture of the Next CERN Accelerator Logging Service." https://databricks.com/blog/2017/12/14/the-architecture-of-the-next-cern-accelerator-logging-service.html, 2017. Last Accessed: 2018-09-30.

[4] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs, "Building a Big Data Platform for Smart Cities: Experience and Lessons from Santander," in *2015 IEEE International Congress on Big Data*, BigDataCongress '15, pp. 592–599, IEEE, 2015.

[5] R. W. Wisniewski, M. Michael, D. Shiloach, and J. E. Moreira, "Scale-up x Scale-out: A Case Study using Nutch/Lucene," in *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*, IPDPS' 07, pp. 1–8, IEEE, 2007.

[6] "Apache Hadoop." https://hadoop.apache.org/. Last Accessed: 2018-09-30.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pp. 1–10, IEEE, 2010.

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 307–320, USENIX Association, 2006.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '10, pp. 10–10, USENIX Association, 2010.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28–38, 2015.

[11] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pp. 5:1–5:16, ACM, 2013.

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, pp. 295–308, USENIX Association, 2011.

[13] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pp. 285–300, USENIX Association, 2014.

[14] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pp. 7:1–7:13, ACM, 2012.

[15] K. Kambatla, V. Yarlagadda, I. Goiri, and A. Grama, "UBIS: Utilization-Aware Cluster Scheduling," in *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium*, IPDPS' 18, pp. 358–367, IEEE, 2018.

[16] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing Data Parallel Computing," in *In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pp. 281–294, USENIX Association, 2012.

[17] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards Automated SLOs for Enterprise Clusters," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pp. 117–134, USENIX Association, 2016.

[18] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin, "Fast Data in the Era of Big Data: Twitter's Real-time Related Query Suggestion Architecture," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 1147–1158, ACM, 2013.

[19] M. Schwarzkopf, "Cluster Scheduling for Data Centers," *ACM Queue*, vol. 15, no. 5, p. 70, 2017.

[20] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-aware Cluster Management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pp. 127–144, ACM, 2014.

[21] Ernst & Young Law, "Data Retention and Preservation - Overview on Requirements in Selected Countries." `https://eylaw.ey.com/2015/07/17/data-retention-and-preservation-in-selected-countries/`, 2015. Last Accessed: 2018-09-30.

[22] S. Childs, J. McLeod, E. Lomas, and G. Cook, "Opening Research Data: Issues and Opportunities," *Records Management Journal*, vol. 24, no. 2, pp. 142–162, 2014.

[23] B. Lee, A. Awad, and M. Awad, "Towards Secure Provenance in the Cloud: A Survey," in *Proceedings of the 8th International Conference on Utility and Cloud Computing*, UCC' 15, pp. 577–582, IEEE, 2015.

[24] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI '04, pp. 10–10, USENIX Association, 2004.

[25] F. Hüske, "Peeking Into Apache Flink's Engine Room." `https://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html/`, 2015. Last Accessed: 2018-09-30.

[26] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning Fast Iterative Data Flows," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1268–1279, 2012.

[27] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand, "Firmament: Fast, Centralized Cluster Scheduling at Scale," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pp. 99–115, USENIX Association, 2016.

[28] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pp. 293–307, 2015.

[29] L. Thamsen, B. Rabier, F. Schmidt, T. Renner, and O. Kao, "Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference.," in *2017 IEEE International Congress on Big Data*, BigData-Congress '17, pp. 145–152, IEEE, 2017.

[30] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, pp. 323–336, USENIX Association, 2011.

[31] M. Veiga Neves, C. De Rose, K. Katrinis, and H. Franke, "Pythia: Faster Big Data in Motion Through Predictive Software-Defined Network Optimization at Runtime," in *Parallel and Distributed Processing Symposium, Proceedings of the 28th International Conference on Parallel and Distributed Processing*, IPDPS '14, pp. 82–90, 2014.

[32] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pp. 265–278, ACM, April 2010.

[33] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-Based Scheduling: If You're Late Don't Blame Us!," in *Proceedings of the 14th ACM Symposium on Cloud Computing*, SOCC '14, pp. 1–14, ACM, 2014.

[34] "Kubernetes - Production-Grade Container Orchestration." `https://kubernetes.io/`. Last Accessed: 2018-09-30.

[35] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-Scale Cluster Management at Google With Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pp. 18:1–18:17, ACM, 2015.

[36] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 77–88, ACM, 2013.

[37] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pp. 351–364, ACM, 2013.

[38] Z. Guo, G. Fox, and M. Zhou, "Investigation of Data Locality in MapReduce," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '12, pp. 419–426, IEEE, 2012.

[39] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *Proceedings of the 22nd Symposium on Operating Systems Principles*, OSDI' 09, pp. 261–276, ACM, 2009.

[40] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards Predictable Datacenter Networks," in *Proceedings of SIGCOMM 2011*, vol. 41 of *SIGCOMM '11*, pp. 242–253, 2011.

[41] S. Moon, J. Lee, and Y. S. Kee, "Introducing SSDs to the Hadoop MapReduce Framework," in *Proceedings of the 7th International Conference on Cloud Computing*, CloudCom '14, pp. 272–279, 2014.

[42] K. Kambatla and Y. Chen, "The Truth About MapReduce Performance on SSDs," in *Proceedings of the 28th Large Installation System Administration Conference*, LISA14 '14, pp. 118–126, USENIX Association, 2014.

[43] K. Krish, M. S. Iqbal, and A. R. Butt, "Venu: Orchestrating Ssds in Hadoop Storage," in *Proceedings of the 2014 IEEE International Conference On Big Data*, BigData '14, pp. 207–212, IEEE, 2014.

[44] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pp. 1–15, ACM, 2014.

[45] S. Nakamoto, "Bitcoin: A Peer-To-Peer Electronic Cash System." `http://www.bitcoin.org/bitcoin.pdf`, 2008. Last Accessed: 2018-09-30.

[46] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger." `http://gavwood.com/Paper.pdf`, 2014. Last Accessed: 2018-09-30.

[47] Hyperledge, "Hyperledger Architecture, Volume 2: Smart Contracts." `https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf`. Last Accessed: 2018-09-30.

[48] R. C. Merkle, "Protocols for Public Key Cryptosystems," in *IEEE Symposium On Security and Privacy*, pp. 122–122, IEEE, 1980.

[49] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts SoK," in *Proceedings of the 6th International Conference on Principles of Security and Trust*, POST '17, pp. 164–186, Springer, 2017.

[50] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *VLDB*, vol. 3, no. 1-2, pp. 285–296, 2010.

[51] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pp. 810–818, ACM, 2010.

[52] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A Warehousing Solution over a Map-Reduce Framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[53] D. Warneke and O. Kao, "Nephele: Efficient Parallel Data Processing in the Cloud," in *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pp. 8:1–8:10, ACM, 2009.

[54] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pp. 59–72, ACM, 2007.

[55] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The Stratosphere Platform for Big Data Analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[56] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/-PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pp. 119–130, ACM, 2010.

[57] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann, "Meteor/Sopremo: An Extensible Query Language and Operator Model," in *Proceedings of the International Workshop on End-To-End Management of Big Data*, BigData 2012, pp. 1–10, VLDB Endowment, 2012.

[58] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, "Opening the black boxes in data flow optimization," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1256–1267, 2012.

[59] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann, "SOFA: An Extensible Logical Optimizer for UDF-heavy Data Flows," *Information Systems*, vol. 52, no. C, pp. 96–125, 2015.

[60] L. Thamsen, T. Renner, and O. Kao, "Continuously Improving the Resource Utilization of Iterative Parallel Dataflows," in *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops*, ICDCSW 2016, pp. 1–6, IEEE, 2016.

[61] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pp. 2–2, USENIX Association, 2012.

[62] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and Rich Analytics at Scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 13–24, ACM, 2013.

[63] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pp. 1383–1394, ACM, 2015.

[64] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pp. 599–613, USENIX Association, 2014.

[65] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A Resilient Distributed Graph System on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pp. 2:1–2:6, ACM, 2013.

[66] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 423–438, ACM, 2013.

[67] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-Of-Order Data Processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[68] D. Jackson, Q. Snell, and M. Clement, "Core Algorithms of the Maui Scheduler," in *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 87–102, Springer, 2001.

[69] S. Iqbal, R. Gupta, and Y.-C. Fang, "Planning Considerations for Job Scheduling in HPC Clusters," *Dell Power Solutions*, pp. 133–136, 2005.

[70] "Nomad." `https://www.nomadproject.io/`. Last Accessed: 2018-09-30.

[71] A. Kuzmanovska, R. H. Mak, and D. Epema, "KOALA-F: A Resource Manager for Scheduling Frameworks in Clusters," in *Proceedings of the 16th International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '16, pp. 80–89, IEEE/ACM, 2016.

[72] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pp. 265–283, USENIX Association, 2016.

[73] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th Symposium on Operating Systems Principles*, SOSP '03, pp. 29–43, ACM, 2003.

[74] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases," in *Proceddings of the 15th Conference on File and Storage Technologies*, FAST '17, pp. 89–104, USENIX Association, 2017.

[75] G. Mackey, S. Sehrish, and J. Wang, "Improving Metadata Management for Small Files in HDFS," in *Proceedings of the International Conference on Cluster Computing*, Cluster '09, pp. 1–4, IEEE, 2009.

[76] X. Liu, J. Han, Y. Zhong, C. Han, and X. He, "Implementing WebGIS on Hadoop: A Case Study of Improving Small File I/O Performance on HDFS," in *Proceedings of the International Conference on Cluster Computing*, Cluster '09, pp. 1–8, IEEE, 2009.

[77] C. Yan, T. Li, Y. Huang, and Y. Gan, "HMFS: Efficient Support of Small Files Processing Over HDFS," in *International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '14, pp. 54–67, Springer, 2014.

[78] T. Renner, J. Müller, L. Thamsen, and O. Kao, "Addressing Hadoop's Small File Problem With an Appendable Archive File Format," in *Proceedings of the 2017 Computing Frontiers Conference*, CF '17, pp. 367–372, ACM, 2017.

[79] A. Davies and A. Orsaria, "Scale Out with GlusterFS," *Linux Journal*, vol. 2013, no. 235, 2013.

[80] "Amazon S3." `https://aws.amazon.com/s3/`. Last Accessed: 2018-09-30.

[81] "Apache Flink Documentation: Connectors." `https://ci.apache.org/projects/flink/flink-docs-master/dev/batch/connectors.html`. Last Accessed: 2018-09-30.

[82] "MongoDB." `https://www.mongodb.com/`. Last Accessed: 2018-09-30.

[83] "Google Cloud Storage." `https://cloud.google.com/storage/`. Last Accessed: 2018-09-30.

[84] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 407–420, 2015.

[85] H. Zhang, B. Xu, J. Yan, L. Liu, and H. Ma, "Proactive Data Placement for Surveillance Video Processing in Heterogeneous Cluster," in *Proceedings of the International Conference on Cloud Computing Technology and Science*, CloudCom' 16, pp. 206–213, 2016.

[86] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 575–585, 2011.

[87] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 515–529, 2010.

[88] H. Liu and D. Orban, "Gridbatch: Cloud Computing for Large-Scale Data-Intensive Batch Applications," in *Proceedings of the 8th IEEE International Symposium On Cluster Computing and the Grid*, CCGrid' 08, pp. 295–305, IEEE, 2008.

[89] A. Shanbhag, A. Jindal, Y. Lu, and S. Madden, "A Moeba: A Shape Changing Storage System for Big Data," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1569–1572, 2016.

[90] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden, "AdaptDB: Adaptive Partitioning for Distributed Joins," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 589–600, 2017.

[91] L. Golab, M. Hadjieleftheriou, H. Karloff, and B. Saha, "Distributed Data Placement to Minimize Communication Costs via Graph Partitioning," in *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, SSDBM' 14, p. 20, ACM, 2014.

[92] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters," in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pp. 287–300, ACM, 2011.

[93] Z. Cheng, Z. Luan, Y. Meng, Y. Xu, D. Qian, A. Roy, N. Zhang, and G. Guan, "ERMS: An Elastic Replication Management System for HDFS," in *Proceedings of the IEEE International Conference on Cluster Computing Workshops*, Cluster' 12, pp. 32–40, 2012.

[94] C. L. Abad, Y. Lu, and R. H. Campbell, "DARE: Adaptive Data Replication for Efficient Cluster Scheduling," in *Proceedings of the IEEE International Conference on Cluster Computing*, Cluster' 11, pp. 159–168, IEEE, 2011.

[95] D. M. Bui, S. Hussain, E. N. Huh, and S. Lee, "Adaptive Replication Management in HDFS Based on Supervised Learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 6, pp. 1369–1382, 2016.

[96] H. E. Ciritoglu, T. Saber, T. Buda, J. Murphy, and C. Thorpe, "Towards a Better Replica Management for Hadoop Distributed File System," in *2018 IEEE International Congress on Big Data*, BigDataCongress '18, pp. 1–8, IEEE, 2018.

[97] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: scheduling of long running applications in shared production clusters," in *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, p. 4, ACM, 2018.

[98] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pp. 239–250, ACM, 2015.

[99] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The Power of Choice in Data-Aware Cluster Scheduling," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, USENIX Association, 2014.

[100] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job Scheduling for Multi-User Mapreduce Clusters," *Tech Report of: EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*, 2009.

[101] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving Mapreduce Performance Through Data Placement in Heterogeneous Hadoop Clusters," in *Proceedings of the IEEE International Symposium On Parallel & Distributed Processing, Workshops and Phd Forum*, IPDPSW' 10, pp. 1–9, IEEE, 2010.

[102] X. Zhang, Z. Zhong, S. Feng, B. Tu, and J. Fan, "Improving Data Locality of MapReduce by Scheduling in Homogeneous Computing Environments," in *Proceedings of the 9th International Symposium On Parallel and Distributed Processing With Applications* , ISPA '11', pp. 120–126, IEEE, 2011.

[103] X. Zhang, Y. Feng, S. Feng, J. Fan, and Z. Ming, "An Effective Data Locality Aware Task Scheduling Method for MapReduce Framework in Heterogeneous Environments," in *Proceedings of the 2011 International Conference On Cloud and Service Computing*, CSC '11', pp. 235–242, IEEE, 2011.

[104] X. Bu, J. Rao, and C.-z. Xu, "Interference and Locality-Aware Task Scheduling for MapReduce Applications in Virtual Clusters," in *Proceedings of the 22Nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '13, pp. 227–238, ACM, 2013.

[105] C.-H. Hsu, K. D. Slagter, and Y.-C. Chung, "Locality and Loading Aware Virtual Machine Mapping Techniques for Optimizing Communications in MapReduce Applications," *Future Generation Computer Systems*, vol. 53, pp. 43–54, 2015.

[106] L. Thamsen, I. Verbitskiy, F. Schmidt, T. Renner, and O. Kao, "Selecting Resources for Distributed Dataflow Systems According to Runtime Targets," in *Proceedings of the 35th IEEE International Performance Computing and Communications Conference*, IPCCC '16, pp. 1–8, IEEE, 2016.

[107] J. Koch, L. Thamsen, F. Schmidt, and O. Kao, "SMiPE: Estimating the Progress of Recurring Iterative Distributed Dataflows," in *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '17, pp. 156–163, IEEE, 2017.

[108] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI '16, pp. 363–378, USENIX Association, 2016.

[109] L. Thamsen, I. Verbitskiy, J. Beilharz, T. Renner, A. Polze, and O. Kao, "Ellis: Dynamically Scaling Distributed Dataflows to Meet Runtime Targets," in *Proceedings of the 2017 IEEE 9th International Conference on Cloud Computing Technology and Science*, CloudCom '17, pp. 146–153, IEEE, 2017.

[110] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 607–618, ACM, 2013.

[111] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pp. 450–462, ACM, 2015.

[112] "IXcoin." https://bitcoinscrypt.io/. Last Accessed: 2018-09-30.

[113] "Bitcoin sCrypt." https://bitcoinscrypt.io/. Last Accessed: 2018-09-30.

[114] "Lite Coin." https://litecoin.org/. Last Accessed: 2018-09-30.

[115] S. Ghoshal and G. Paul, "Exploiting Block-Chain Data Structure for Auditorless Auditing on Cloud Data," in *Proceedings of the Internal Conference on Information Systems Security*, ICISS '16, pp. 359–371, Springer, 2016.

[116] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla, "Provchain: A Blockchain-Based Data Provenance Architecture in Cloud Environment With Enhanced Privacy and Availability," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '16, pp. 468–477, IEEE, 2017.

[117] A. Ramachandran and M. Kantarcioglu, "SmartProvenance: A Distributed, Blockchain Based DataProvenance System," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pp. 35–42, ACM, 2018.

[118] M. Ali, J. C. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A Global Naming and Storage System Secured by Blockchains," in *USENIX Annual Technical Conference*, ATC '16', pp. 181–194, 2016.

[119] Filecoin, "A Cryptocurrency Operated File Network." http://filecoin.io/filecoin.pdf, 2014. Last Accessed: 2018-09-30.

[120] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz, "Permacoin: Repurposing Bitcoin Work for Data Preservation," in *Proceedings of the IEEE Symposium on Security and Privacy*, SP '14', pp. 475–490, 2014.

[121] S. Wilkinson, T. Boshevski, J. Brandoff, and V. Buterin, "Storj a Peer-To-Peer Cloud Storage Network." https://storj.io/storj.pdf (accessed November 2017), 2014. Last Accessed: 2018-09-30.

[122] D. Vorick and L. Champine, "Sia: Simple Decentralized Storage." https://www.sia.tech/whitepaper.pdf, 2014. Last Accessed: 2018-09-30.

[123] X. L. Yu, X. Xu, and B. Liu, "EthDrive: A Peer-to-Peer Data Storage with Provenance," in *Proceedings of the 29th International Conference on Advanced Information Systems Engineering*, CAiSE '17, pp. 25–32, 2017.

[124] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System." `https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf`. Last Accessed: 2018-09-30.

[125] B. Antony, "HDFS Storage Efficiency Using Tiered Storage." `https://www.ebayinc.com/stories/blogs/tech/hdfs-storage-efficiency-using-tiered-storage/`, 2015. Last Accessed: 2018-09-30.

[126] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pp. 235–244, ACM, 2011.

[127] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed Job Latency in Data Parallel Clusters," in *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pp. 99–112, ACM, 2012.

[128] T. Renner and L. Thamsen and O. Kao, "CoLoc: Distributed Data and Container Colocation for Data-intensive Applications," in *2016 IEEE International Conference on Big Data*, BigData 2016, pp. 3008–3015, IEEE, 2016.

[129] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez, "Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks," in *Proceedings of the IEEE 2016 International Conference on Cluster Computing*, Cluster 2016, pp. 433–442, IEEE, September 2016.

[130] T. Renner, L. Thamsen, and O. Kao, "Adaptive Resource Management for Distributed Data Analytics Based on Container-level Cluster Monitoring," in *7th International Conference on Data Science, Technology and Applications (DATA)*, pp. 38–47, SCITEPRESS, 2017.

[131] "TPC-H Benchmark." `http://www.tpc.org/tpch/`. Last Accessed: 2018-09-30.

[132] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281–297, University of California Press, 1967.

[133] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient Algorithms for Graph Manipulation," *Communication ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[134] I. Verbitskiy, L. Thamsen, and O. Kao, "When to Use a Distributed Dataflow Engine: Evaluating the Performance of Apache Flink," in *Proceedings of the IEEE International Conference on Cloud and Big Data Computing*, CBDCom 2016, pp. 698–705, IEEE, July 2016.

[135] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?," in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pp. 591–600, ACM, April 2010.

[136] T. Renner, L. Thamsen, and O. Kao, "Network-Aware Resource Management for Scalable Data Analytics Frameworks," in *Proceedings of the 1st First Workshop on Data-Centric Infrastructure for Big Data Science, Co-Located With the 2015 IEEE International Conference on BigData*, DIBS '15, pp. 2793–2800, IEEE, 2015.

[137] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, ACM, November 2013.

[138] C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Transactions on Computers*, vol. 34, no. 10, pp. 892–901, 1985.

[139] H. Alkaff, I. Gupta, and L. Leslie, "Cross-Layer Scheduling in Cloud Systems," in *Proceedings of International Conference On Cloud Engineering*, IC2E '15', pp. 236–245, 2015.

[140] K. Kirkpatrick, "Software-defined networking," *Communications of the ACM*, vol. 56, no. 9, pp. 16–19, 2013.

[141] S. Kirkpatrick, C. Gelatt Jr, and M. Vecchi, "Optimization by simulated annealing," *SCIENCE*, vol. 220, no. 4598, 1983.

[142] T. Renner, J. Müller, and O. Kao, "Endolith: A Blockchain-Based Framework to Enhance Data Retention in Cloud Storages," in *Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '18, pp. 627–634, Euromicro, 2018.

[143] Ethereum White Paper, "A Next-Generation Smart Contract and Decentralized Application Platform." `https://github.com/ethereum/wiki/wiki/White-Paper`. Last Accessed: 2018-09-30.